An Application Program Interface (API)

API components:
- Compiler directives
- Runtime library routines
- Environment variables

Jointly defined and endorsed by a group of major computer hardware and software vendors

What Is OpenMP?
• Meant for distributed memory parallel systems (by itself)
• Necessarily implemented identically by all vendors
• Guaranteed to make the most efficient use of shared memory
• Required to check for data dependencies, data conflicts, race conditions, or deadlocks
• Required to check for code sequences that cause a program to be classified as non-conforming
• Meant to cover compiler-generated automatic parallelization and directives to the compiler to assist such parallelization
• Designed to guarantee that input or output to the same file is synchronous when executed in parallel. The programmer is responsible for synchronizing input and output.

What OpenMP Isn’t

• Standardization:
  ◦ Provide a standard among a variety of shared memory architectures/platforms
• Lean and Mean:
  ◦ 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
• Ease of Use:
  ◦ Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach
  ◦ Provide the capability to implement both coarse-grain and fine-grain parallelism
• Portability:
  ◦ Supports Fortran (77, 90, and 95), C, and C++
  ◦ Public forum for API and membership

Goals of OpenMP
```c
#include <omp.h>
#include <stdio.h>
int main(int argc, char* argv[]) {
    #pragma omp parallel
    printf("Hello, world.\n");
    return 0;
}
```

**Hello world**

```
mpicc -fopenmp -o test test.c
```

**Hello World - continue**
Extensions

Extensions

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

int main() {
    #pragma omp parallel
    printf("Hello, world! This is thread \%d of \%d\n",
           omp_get_thread_num(), omp_get_num_threads());
}
```
mpicc -fopenmp -o test1 test1.c

Compile

- Shared memory, thread based parallelism
- Explicit parallelism
- Fork-Join model

OpenMP Programming Model

- Compiler directive based
- Supports nested parallelism
- Dynamic threads
- I/O – up to the user to get it right
```c
#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 . . .
}
```

**C/C++ Code Structure**

- Case sensitive
- Directives follow conventions of the C/C++ standards for compiler directives
- 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

**General Rules for Pragmas**
• When a thread reaches a **parallel** directive, it creates a team of threads and becomes the master of the team
  ◦ The master is a member of that team and has thread number 0 within that team
• Starting from the beginning of this parallel region, the code is duplicated and all threads execute that code
• There is an implied barrier at the end of a parallel section
  ◦ Only the master thread continues execution past this point
• If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.

```c
#pragma omp parallel ...
#include <omp.h>
int 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 */
} /* main */
```

**Parallel Region Example**
A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it. Work-sharing constructs do not launch new threads. There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct.

**Work-Sharing constructs**

- Shares iterations of a loop across the team
  - Represents a type of "data parallelism"
• Breaks work into separate, discrete sections
  ◦ Each section is executed by a thread.
  ◦ Can be used to implement a type of "functional parallelism"

• Serializes a section of code
A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.

Work-sharing constructs must be encountered by all members of a team or none at all.

Successive work-sharing constructs must be encountered in the same order by all members of a team.

**Restrictions**

- Simple vector-add program
  - Arrays A, B, C, and variable N will be shared by all threads.
  - Variable I will be private to each thread; each thread will have its own unique copy.
  - The iterations of the loop will be distributed dynamically in CHUNK sized pieces.
  - Threads will not synchronize upon completing their individual pieces of work (NOWAIT).

**Example: for directive**
```c
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000
int main ()
{  int i, chunk;
   float a[N], b[N], c[N];
   /* Some initializations */
   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 */
}
```

**Example: for directive**

- Simple program demonstrating that different blocks of work will be done by different threads.

**Example: SECTIONS Directive**
```c
#include <omp.h>
#define N 1000
int main ()
{
    int i;
    float a[N], b[N], c[N], d[N];
    /* Some initializations */
    for (i=0; i < N; i++)
    {
        a[i] = i * 1.5;
        b[i] = i + 22.35;
    }
    #pragma omp parallel shared(a,b,c,d) private(i)
    {
        #pragma omp sections nowait
        {
            #pragma omp section
            for (i=0; i < N; i++)
            {
                c[i] = a[i] + b[i];
            }
            #pragma omp section
            for (i=0; i < N; i++)
            {
                d[i] = a[i] * b[i];
            }
        }
    }
}
```

**Example: SECTIONS Directive**

- The MASTER directive specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code
- `#pragma omp master structured_block`

**Synchronization Constructs**
The CRITICAL directive specifies a region of code that must be executed by only one thread at a time.

```
#pragma omp critical [ name ] newline
structured_block
```

**Critical Directive**

```c
#include <omp.h>
int main()
{
  int x;
  x = 0;
  #pragma omp parallel shared(x)
  {
    #pragma omp critical
    x = x + 1;
  } /* end of parallel section */
}
```

**Critical Example**
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.

```c
#pragma omp barrier
```

**Barrier Directive**

- [https://computing.llnl.gov/tutorials/openMP](https://computing.llnl.gov/tutorials/openMP)

**Tutorials**