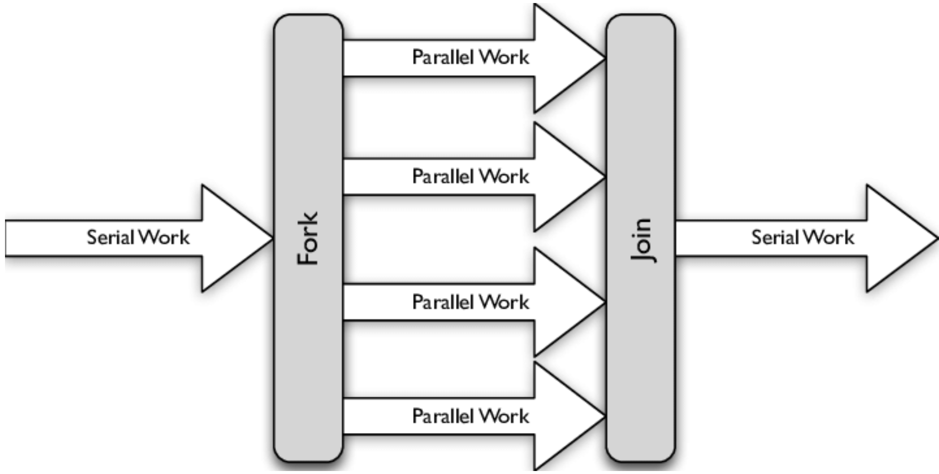
Introduction to OpenMP

OpenMP API

- **Directives** and **clauses** to specify the parallelism, synchronization, variable sharing types (private, shared, …), …
- **Library functions** for certain functionalities in runtime
  - Modifying number of threads or scheduling policies **in runtime**
  - Getting current number of threads or scheduling policies, etc.
- **Environment variables** to modify code behavior **without recompiling**
  - Number of threads (OMP_NUM_THREADS=??)
  - Scheduling policies (OMP_SCHEDULE=??)
  - To specify during the code execution (e.g., OMP_NUM_THREADS=4 ./exec)

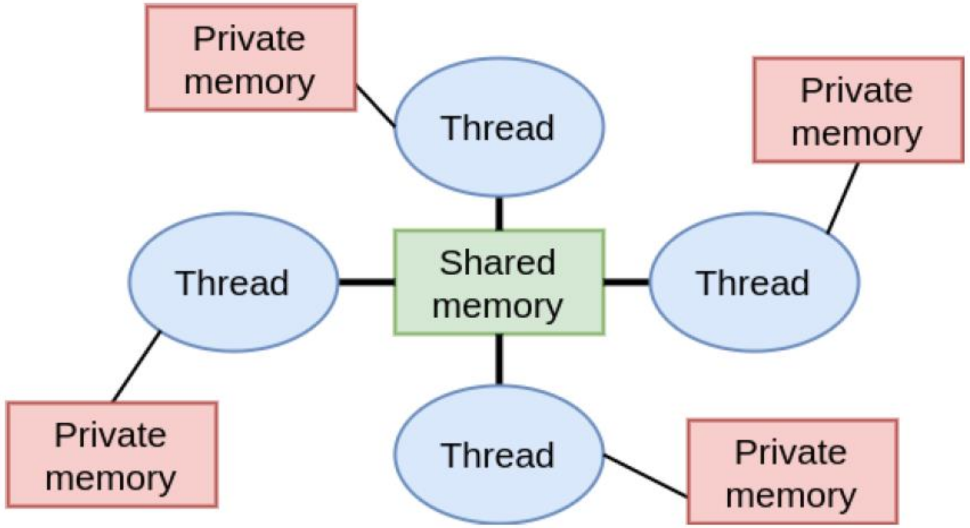
OpenMP execution model

- The programmer adds directives that create **parallel regions** on a code block
  - Multiple threads are created for this code block
  - Each thread executes **the entire code block**, but with a different thread identifier
  - Threads work **asynchronously** (can execute different lines of the code block) and synchronize at the end of the parallel region
  - **Work sharing** should be performed (otherwise same computation would be done redundantly)
  - Thread creation roughly takes 10-20ms.
- At the end of the parallel region, all threads except the master (thread 0) are **destroyed**
- Master thread then continues the sequential execution until the next parallel region or the end of the program
- Data (variables, arrays, …) belonging to the master thread can be made available to other threads



OpenMP memory model

- All threads have access to the same **shared memory space**
  - Variables can be **shared** and **accessed** by all threads
  - Each thread can still have a **private memory and variables**
  - Memory transfers are transparent to the programmer (handled automatically)



Example: Vector inner product (a^Tb)

```
#include <stdio.h>
#define SIZE 256
int main() {
  int i;
  double innerp, a[SIZE], b[SIZE];
  // Initialization
  sum = 0.;
  for (i = 0; i < SIZE; i++) {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
  }
  // Computation
  for (i = 0; i < SIZE; i++) { innerp = innerp + a[i] * b[i]; }
  printf("inner product = %lf\n", innerp);
  return 0;
}
```

Example: Vector inner product using OpenMP

```
#include <stdio.h>
#include "omp.h"
#define SIZE 256
int main() {
  int i;
  double innerp, a[SIZE], b[SIZE];
  // Initialization
  innerp = 0.;
  for (i = 0; i < SIZE; i++) {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
  }
  // Computation
#pragma omp parallel for reduction(+:innerp)
  for (i = 0; i < SIZE; i++) { innerp = innerp + a[i] * b[i]; }
  printf("innerp = %g\n", sum);
  return 0;
}
```

Compiling and executing an OpenMP program

- Compilation: g++ program.cpp -o program –**fopenmp**

- Execution: ./program
    - Alternatively to execute using X threads: OMP_NUM_THREADS=X ./program

OpenMP directives

Thread creation and basic management

OpenMP directives (**#pragma omp** *directive*)

- A *directive* is a hint to the compiler to perform a code transformation.
- Creating a parallel region (i.e., creating threads)
  - **parallel**
- Sharing work (not re-doing by each thread) among threads **within a parallel region**
  - **sections:** defining code blocks that can be executed independently
  - **for:** sharing the iterations of a loop among threads
  - **single:** defining a code block to be executed by a single thread only
  - **master:** defining a code block to be executed by the master thread
- Synchronization/coordination
  - **critical**: defining a code block to be executed by **one thread at a time**
  - **atomic:** performing atomic instructions (+=, -=, *=, …) on a single variable
  - **barrier:** adding a synchronization point for all threads in a parallel region

omp parallel directive

```
#pragma omp parallel default(none) num_threads(P) [clause1 clause2 …]
{
  // Parallel code to be executed by each of P threads created
}
```

- Creates a parallel region having P threads (P can be constant/variable)
- Each thread executes the entire code block line by line
- Threads are asynchronous by default (can execute different lines)
- If **num_threads** not specified, following #threads will be used instead:
    - value set by **omp_set_num_threads(P)** function in omp.h
    - value set by **OMP_NUM_THREADS** environment variable
    - #threads supported in the hardware (typically #cores x 2 for a CPU with SMT)

Example: Printing "oh, no no no!" using three threads

- openmp-example-1.cpp
- The master thread prints "oh, "
- Create three threads within a parallel region, each printing " no"
- After the parallel region, the master thread prints "!"

Thread identifier

```
#pragma omp parallel default(none) num_threads(P)
{
  // Parallel code to be executed by each of P threads created
  int thid = omp_get_thread_num();
  int numth = omp_get_num_threads();
}
```

- **omp_get_thread_num()** gives the identifier of a thread
  - Must be called within a parallel region; otherwise it gives 0
  - Must use a **variable private to a thread** to store it
- **omp_get_num_threads()** gives the number of threads available currently
  - Must be called within a parallel region; otherwise it gives 1
  - A **shared variable** might still be OK to store it.
- **thid** and **numth** can be used to differentiate/distribute work among threads

Variable types: **shared**

- Variables defined within the parallel region remain private to each thread and invisible to others
- Private variables are destroyed at the end of a parallel region
- By default, all variables of the master thread (defined before the parallel region) are shared/visible to all threads.
- **default(none)** clause makes these variables invisible, and demands explicit sharing with **shared(varName)** clause
  - Good practice to use this, prevents bugs!

```
int x = 3;                          // x is shared by all threads

#pragma omp parallel num_threads(P)
{
  int y = x + omp_get_thread_num(); // each th has its own
}
```

Variable types: **shared**

- Variables defined within the parallel region remain private to each thread and invisible to others
- Private variables are destroyed at the end of a parallel region
- By default, all variables of the master thread (defined before the parallel region) are shared/visible to all threads.
- **default(none)** clause makes these variables invisible, and demands explicit sharing with **shared(varName)** clause
  - Good practice to use this, prevents bugs!

```
int x = 3;                          // x is not visible to threads

#pragma omp parallel default(none) num_threads(P)
{
  int y = x + omp_get_thread_num(); // each th has its own
}
```

Compilation error! x is undefined

Variable types: **shared**

- Variables defined within the parallel region remain private to each thread and invisible to others
- Private variables are destroyed at the end of a parallel region
- By default, all variables of the master thread (defined before the parallel region) are shared/visible to all threads.
- **default(none)** clause makes these variables invisible, and demands explicit sharing with **shared(varName)** clause
  - Good practice to use this, prevents bugs!

```
int x = 3;                         // x is visible to threads

#pragma omp parallel default(none) num_threads(P)
shared(x)
{
  int y = x + omp_get_thread_num(); // each th has its own
}
```

x is visible again.

Variable types: **shared**

- Reading a shared variable simultaneously in different threads poses no problem
- Modifying a shared variable can create conflicts called a **race condition**.
  - It requires handling write/write or write/read conflicts.
  - **atomic** or **critical** constructs can be used

```
int x = 3;                    // x is visible to threads

#pragma omp parallel default(none) num_threads(P)
shared(x)
{
  x = x + 1;
}
```

Example: Computing "nine" using three threads

- openmp-example-2.cpp
- Three functions (computeTwo(), computeThree(), computeFour()) are given, which take 2, 3, and 4 seconds to return the values of 2, 3, and 4, respectively.
- Write an OpenMP program that computes 2, 3, and 4 in parallel using these functions, then adds them together to compute 9.

omp atomic directive

- When modifying a shared variable by multiple threads simultenously, the result can be wrong due to a **race condition.**
- **atomic** directive calls a hardware instruction for simple arithmetic/logic operations (+,-,*,min,max,and,…) that carry out three subinstructions in a single shot
    - load(x)
    - add(x, 1)
    - store(x, add(x, 1))
- Prevents race conditions, but is not cheap
    - Should minimize its use (particularly in a loop)
    - If there are multiple contributions by a thread, accumulate them in a private variable first, then add to the shared variable with **atomic** operation

```
int x = 3;                    // x is visible to threads

#pragma omp parallel default(none) num_threads(P)
shared(x)
{
#pragma omp atomic
  x = x + 1;
}
```

OpenMP directives

Work-sharing constructs, loop scheduling, barriers

omp sections directive

- Creates independent code blocks or **sections**
- Must be done **within a parallel region**
- Each **section** is a parallel task, and is executed by **only one thread** (instead of each thread)
- Provides static parallelism (since the number of sections is fixed in the code)
- Can have more/less sections than #threads available; task distribution is handled by OpenMP
- **OpenMP Tasks** provide a more flexible framework (we will see later)

```
#pragma omp parallel default(none) num_threads(P)
{
#pragma omp sections
  {
#pragma omp section
    {
      f();
    } // end of section
#pragma omp section
    {
      g();
    } // end of section
    …
  } // end of sections, implicit barrier for all threads
}
```

Example: Computing "nine" using **sections**

- openmp-example-3.cpp
- Three functions (computeTwo(), computeThree(), computeFour()) are given, which take 2, 3, and 4 seconds to return the values of 2, 3, and 4, respectively.
- Write an OpenMP program that computes 2, 3, and 4 in parallel using a **section** for each
- Next, adds them together within a shared variable using **atomic** to compute 9

omp single/master directive

- **omp single** creates a sequential region within a parallel region; the code block is executed by a single thread (first thread available)
- **omp master** does the same, but the code block is executed by the **master thread** (i.e., thread 0)
- There is an implicit barrier after **omp single**, and no barrier after **omp master**
- Useful for not having to close and reopen a parallel region for a sequential computation, avoiding thread creation/destruction overhead

```
#pragma omp parallel default(none) num_threads(P)
{
#pragma omp single          // executed by 1 thread
  {
    f();
  } // end of single, implicit barrier for all threads
#pragma omp master          // executed by master thread
  {
    g();
  } // end of master, no barrier, others continue the rest
#pragma omp single          // executed by 1 thread
  {
    h();
  } // end of single, implicit barrier for all threads
}
```

omp critical directive

- **omp critical** creates a region within a parallel region; the code block is executed by all threads yet a single thread at a time
- Prevents data conflicts / race conditions
- No implicit barrier at the end
- Useful for non-trivial operations for which **atomic** is not provided

```
#pragma omp parallel default(none) num_threads(P)
{
  int thid = omp_get_thread_num();
#pragma omp critical            // executed by all threads
  {                             // but one thread at a time
    nonthreadsafe_function(thid);
  } // end of critical, no implicit barrier
}
```

omp for directive

- **omp for** distributes the domain of iteration of a for loop among threads, instead of repeating the entire loop at each thread.
- Each loop iteration is executed only once by one of threads
- There is an implicit barrier after **omp for**
- Distribution of iteration depends on the **scheduling policy** and **chunk size** of distribution
    - By default, each thread gets N/P contiguous iterations

```
int N = …;
#pragma omp parallel default(none) num_threads(P)
shared(N)
{
#pragma omp for
  for (int i = 0; i < N; i++) {
    f(i);
  } // end of for, implicit barrier
}
```

Example: Initializing an array

- openmp-example-4.cpp
- Allocate an array A with N integers
- Initialize each element A[i] to i, for 1 <= i <= N
- Use two **omp sections** to parallelize the loop
- What would happen if we decide to use more/less sections/threads?

Example: Initializing an array

- openmp-example-5.cpp
- Allocate an array A with N integers
- Initialize each element A[i] to i, for 1 <= i <= N
- Use **omp for** to parallelize the loop
- What would happen if we decide to use more/less threads?