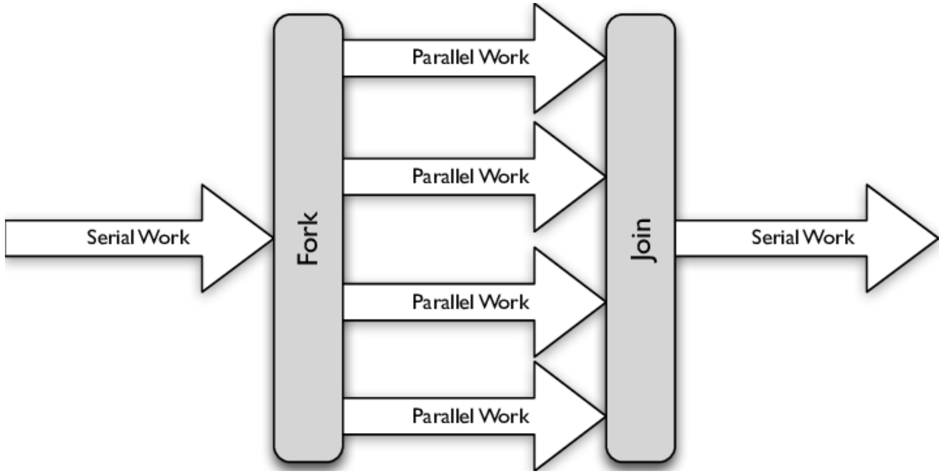
Introduction to OpenMP

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)

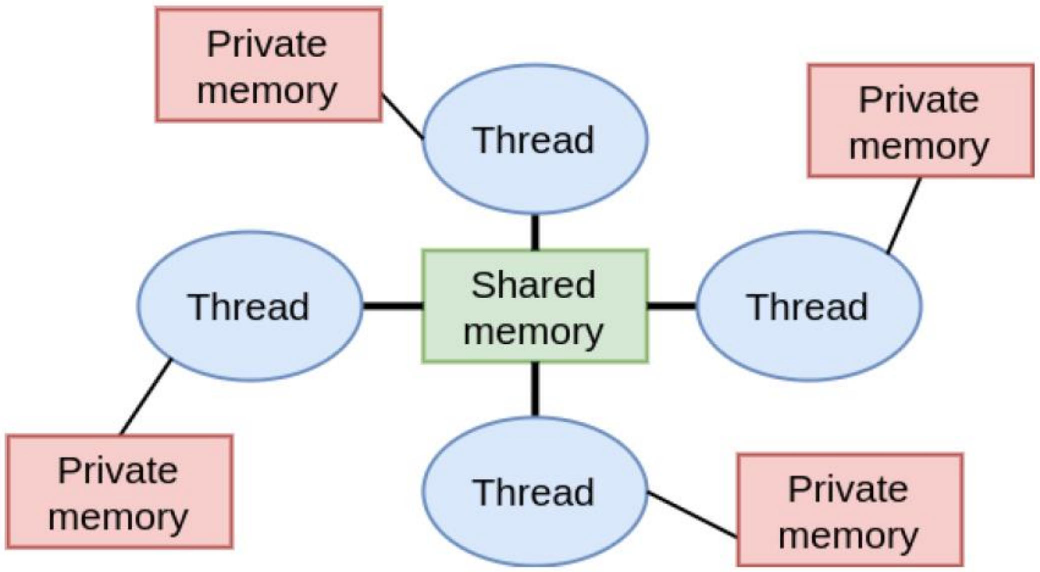
OpenMP execution model

- 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 id
  - **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



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 using OpenMP

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

OpenMP directives

Thread creation and basic management

OpenMP directives (**#pragma omp …**)

- Creating a parallel region
  - **parallel**
- Sharing work (not re-doing at each thread) **within a parallel region**
  - **for:** sharing the iterations of a loop among threads
  - **sections:** defining code blocks that can be executed independently
  - **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 num_threads(P) [clause1 clause2 …]
{
  // Parallel code to be executed by each thread
}
```

- 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 if CPU with SMT)

Thread identifiers

```
#pragma omp parallel num_threads(4)
{
  // Parallel code to be executed by each thread
  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 **private variable** 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** is still OK to store it.
- **thid** and **numth** can be used to differentiate/distribute work among threads

if clause

```
#pragma omp parallel num_threads(P) if (cond)
{
  // Parallel code to be executed by each thread
}
```

- **if** clause can be added when creating a parallel region
- Threads are created only if **cond** is true/nonzero, otherwise only the master thread executes the block of code
- Useful for preventing thread creation overhead for small problems

Variable types

- By default, all variables defined before the parallel region are shared/visible to all threads
- Each variable defined within the parallel region is private to each thread, and are not visible to others
- Private variables are destroyed at the end of a parallel region

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

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

OpenMP directives

Work-sharing constructs, loop scheduling

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
- Can have more/less sections than #threads available; task distribution is handled by OpenMP
- **OpenMP Tasks** provides a more flexible framework

```
#pragma omp parallel 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
}
```

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**
- 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, avoiding thread creation/destruction overhead

```
#pragma omp parallel num_threads(P)
{
#pragma omp single           // executed by 1 thread
  {
    f()
  } // end of single, implicit barrier
#pragma omp master           // executed by master thread
  {
    g();
  } // end of master, no barrier
#pragma omp single           // executed by 1 thread
  {
    h();
  } // end of single, 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

```
#pragma omp parallel num_threads(P)
{
#pragma omp for
  for (int i = 0; i < N; i++)
  {
    f(i);
  } // end of for, implicit barrier
}
```

schedule clause for **omp for**

- Determines how iterations are distributed among threads
- **schedule(policy, chunksize)**
- **static** policy: Assign chunksize (default=N/num_threads) contiguous iterations to each thread in a circular order (0,1,2,3,0,1,2,3,…)
- **dynamic** policy: Assign chunksize (default=1) contiguous iterations to the first available thread (0,3,2,0,1,2,1,3,2,0,…)
- **guided** policy: Start assigning big chunks dynamically, gradually reduce it to chunksize towards the end.
- **runtime** policy: Defer the decision to user in runtime (e.g. using OMP_SCHEDULE variable)

```
#pragma omp parallel num_threads(P)
{
#pragma omp for schedule(dynamic, 32)
  for (int i = 0; i < N; i++)
  {
    f(i);
  } // end of for, implicit barrier
#pragma omp for schedule(static)
  for (int i = 0; i < N; i++)
  {
    g(i);
  } // end of for, implicit barrier
}
```