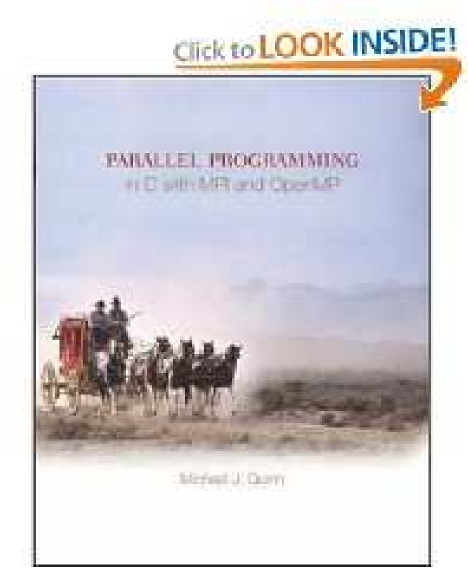
OpenMP programming

Overview

- Basic use of OpenMP: API for shared-memory parallel programming
- Chapter 17 in *Michael J. Quinn*, **Parallel Programming in C with MPI and OpenMP**



Thread programming for shared memory

- Thread programming is a natural model for shared memory
  - Execution unit: thread
  - Many threads have access to shared variables
  - Information exchange is (implicitly) through the shared variables
- Several thread-based programming environments
  - Pthreads
  - Java threads
  - Intel Threading Building Blocks (TBB)
  - OpenMP

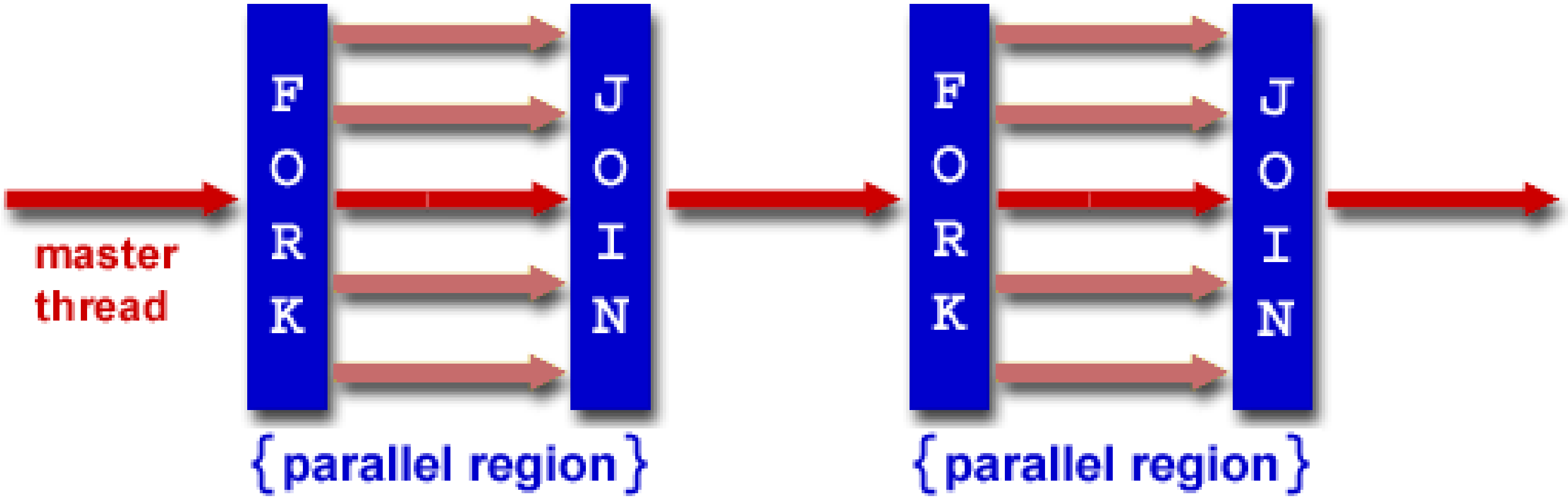
OpenMP

- OpenMP is a portable standard for shared-memory programming
- The OpenMP API consists of
  - compiler directives
  - library routines
  - environment variables
- Advantages:
  - User-friendly
  - Incremental parallelization of a serial code
  - Possible to have a single source code for both serial and parallelized versions
- Disadvantages:
  - Relatively limited user control
  - Most suitable for parallelizing loops (data parallelism)
  - Performance?

The programming model of OpenMP

- Multiple cooperating threads are allowed to run simultaneously
- Threads are created and destroyed dynamically in a **fork-join** pattern
  - An OpenMP program consists of a number of parallel regions
  - Between two parallel regions there is only one master thread
  - In the beginning of a parallel region, a team of new threads is spawned
  - The newly spawned threads work simultaneouly with the master thread
  - At the end of a parallel region, the new threads are destroyed

Fork-join model



`https://computing.llnl.gov/tutorials/openMP/`

The memory model of OpenMP

- Most variables are shared between the threads
- Each thread has the possibility of having some private variables
  - Avoid race conditions
  - Passing values beween the sequential part and the parallel region

OpenMP: first things first

- Always remember the header file `#include <omp.h>`
- Insert compiler directives (`#pragma omp...`), possibly also some OpenMP library routines
- Compile
  - For example, `gcc -fopenmp code.c`
- Assign the environment variable `OMP_NUM_THREADS`
  - It specifies the total number of threads inside a parallel region, if not otherwise overwritten
  - For example, in connection with submitting a batch job, it is often necessary to modify the `.bashrc` file:

    ```
    export OMP_NUM_THREADS=x
    ```

General code structure

```
#include <omp.h>

main ()  {

  int var1, var2, var3;

  /* serial code */
  /* ... */

  /* start of a parallel region */
#pragma omp parallel private(var1, var2) shared(var3)
  {
    /* ... */
  }

  /* more serial code */
  /* ... */

  /* another parallel region */
  /* ... */
}
```

Important library routines

- `int omp_get_num_threads ();`
  returns the number of threads inside a parallel region
- `int omp_get_thread_num ();`
  returns the “thread id” for each thread inside a parallel region

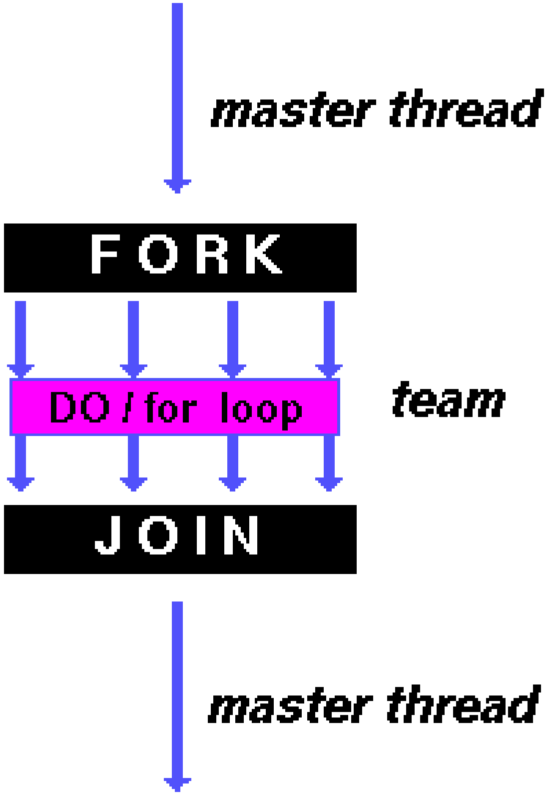
Parallel region

- The following compiler directive creates a parallel region
  `#pragma omp parallel { ...  }`
- Clauses can be added at the end of the directive
- Most often used clauses:
  - `default(shared)` or `default(none)`
  - `public(list_of_variables)`
  - `private(list_of_variables)`

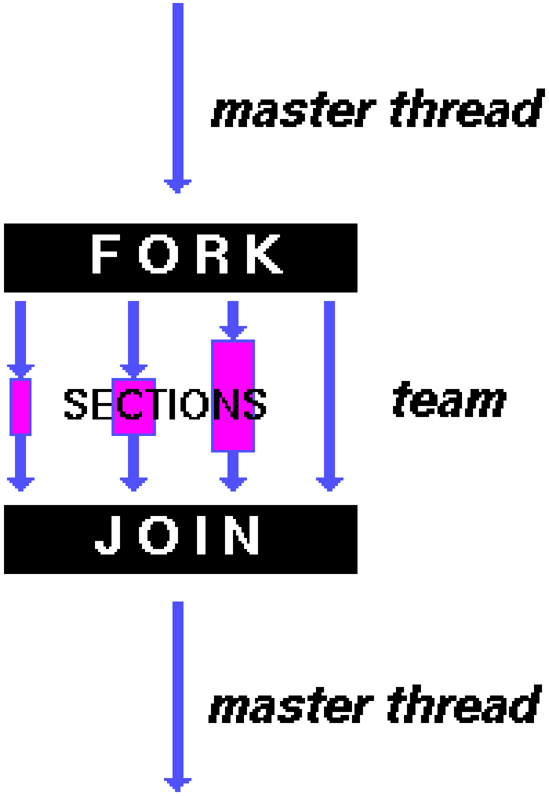
Hello-world in OpenMP

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

int main (int argc, char *argv[])
{
  int th_id, nthreads;

  #pragma omp parallel private(th_id)
  {
    th_id = omp_get_thread_num();
    printf("Hello World from thread %d\n", th_id);

    #pragma omp barrier

    if ( th_id == 0 ) {
      nthreads = omp_get_num_threads();
      printf("There are %d threads\n",nthreads);
    }
  }

  return 0;
}
```

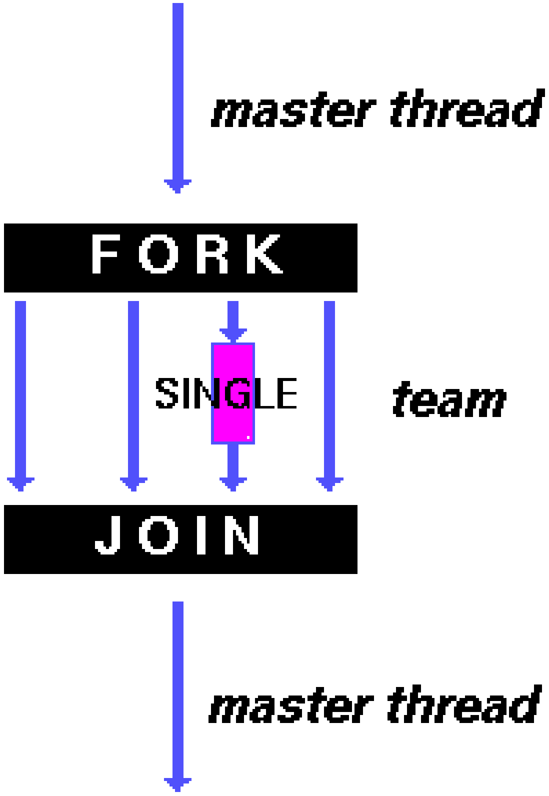
Work-sharing contructs



`omp for`



`omp sections`



`omp single`

https://computing.llnl.gov/tutorials/openMP/

Parallel for loop

- Inside a parallel region, the following compiler directive can be used to parallelize a `for`-loop:
  `#pragma omp for`
- Clauses can be added at the end of the directive
  - `schedule(static, chunk_size)`
  - `schedule(dynamic, chunk_size)`
  - `schedule(guided, chunk_size)`
  - `schedule(auto)`
  - `schedule(runtime)`
  - `private(list_of_variables)`
  - `reduction(operator:variable)`
  - `nowait`

Example

```
#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)
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];
  }  /* end of parallel region */
}
```

More about parallel for

- The number of loop iterations can not be non-deterministic
  - `break`, `return`, `exit`, `goto` not allowed inside the `for`-loop
- The loop index is private to each thread
- The reduction variable is special
  - During the `for`-loop there is a local private copy in each thread
  - At the end of the `for`-loop, all the local copies are combined together by the reduction operation
- Unless the `nowait` clause is used, an implicit barrier synchronization will be added at the end by the compiler
- `#pragma omp parallel` and `#pragma omp for` can be combined into
  `#pragma omp parallel for`

Example of computing inner-product

```
  int i;
  double sum = 0.;
  ...

#pragma omp parallel for default(shared) private(i) reduction(+:sum)
  for (i=0; i<length; i++)
    sum += a[i]*b[i];
}
```

Parallel sections

Inside a parallel region:

```
#pragma omp parallel sections
{
#pragma omp section
  ...
#pragma omp section
  ...
#pragma omp section
  ...
}
```

Example

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

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

  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
    {
    #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];
    }  /* end of sections */
  }  /* end of parallel region */
}
```

Single execution

- `#pragma omp single { ... }`
- `#pragma omp master { ... }`

Coordination and synchronization

- `#pragma omp critical` { `block of codes` }
- `#pragma omp atomic` { `only one statement` }
- `#pragma omp barrier`

Data scope

- OpenMP data scope attribute clauses:
  - `private`
  - `firstprivate`
  - `lastprivate`
  - `shared`
  - `reduction`
- Purposes:
  - define how and which variables are transferred to a parallel region (and back)
  - define which variables are visible to all threads in a parallel region, and which variables are privately allocated to each thread

Some remarks

- When entering a parallel region, the `private` clause ensures each thread having its own new variable instances. The new variables are assumed to be uninitialized.
- A shared variable exists in only one memory location and all threads can read and write to that address. It's the programmer's responsibility to ensure that multiple threads properly access a shared variable.
- The `firstprivate` clause combines the behavior of the `private` clause with automatic initialization.
- The `lastprivate` clause combines the behavior of the `private` clause with a copy back (from the last loop iteration or section) to the original variable outside the parallel region.

Parallelizing nested for loops

- Serial code

  ```
  for (i=0; i<100; i++)
    for (j=0; j<100; j++)
      a[i][j] = b[i][j] + c[i][j]
  ```

- Parallelization

  ```
  #pragma omp parallel for private(j)
  for (i=0; i<100; i++)
    for (j=0; j<100; j++)
      a[i][j] = b[i][j] + c[i][j]
  ```

- Why not parallelize the inner loop?
  - to save overhead of repeated thread forks-joins
- Why must `j` be `private`?
  - to avoid race condition among the threads

Exercises

- Exercise 17.2 from the textbook
- Exercise 17.3 from the textbook
- Write a simple C code to compute the inner-product of two very long vectors. Use `#pragma omp parallel for` to do the parallelization. Choose different schedulers and chunksizes and observe the time usage.