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
What is OpenMP?

- OpenMP is a portable standard for shared-memory programming
- The OpenMP API consists of
  - compiler directives (for insertion into sequential Fortran/C/C++ code)
  - a few library routines
  - some 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 simultaneously 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 between 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
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.
- `void omp_set_num_threads(int)` sets the number of threads to be used.
Parallel region

The following compiler directive creates a parallel region

```c
#pragma omp parallel {
  ...
}
```

Clauses can be added at the end of the directive

Most often used clauses:

- `default(shared)` or `default(none)`
- `shared(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 constructs

[Diagram showing work-sharing constructs with `fork` and `join` keywords, and `do`/`for` loop, `sections`, and `single` keywords]

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:

```c
#pragma omp for
```

Clauses can be added, such as

- `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

```c
#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
- A 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

\[ \sum_{i=0}^{N-1} a_i b_i = a_0 b_0 + a_1 b_1 + \ldots + a_{N-1} b_{N-1} \]

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

/* allocating and initializing arrays 'a' 'b' */
/* ... */

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

Different threads do different tasks independently, each section is executed by one thread.

```c
#pragma omp parallel sections
{
    #pragma omp section
    funcA ();

    #pragma omp section
    funcB ();

    #pragma omp section
    funcC ();
}
```
#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 { ... }
  - code executed by one thread only, no guarantee which thread
  - an implicit barrier at the end

- #pragma omp master { ... }
  - code executed by the master thread, guaranteed
  - no implicit barrier at the end
Coordination and synchronization

- `#pragma omp barrier`
  - synchronization, must be encountered by all threads in a team (or none)

- `#pragma omp ordered { a block of codes }
  - another form of synchronization (in sequential order)

- `#pragma omp critical { a block of codes }
  - more efficient than `#pragma omp critical`

- `#pragma omp atomic { single assignment statement }`
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

Comments
  OpenMP 2.5 allows parallelization of only one loop layer
  OpenMP 3.x has a new \texttt{collapse} clause
Nested parallelism

When a thread in a parallel region encounters another parallel construct, it may create a new team of threads and become the master of the new team.

```c
#pragma omp parallel num_threads(4)
{
    /* .... */

    #pragma omp parallel num_threads(2)
    {
        /* .... */
    }
}
```
Starting with OpenMP 3.0: #pragma omp task

#pragma omp parallel shared(p_vec) private(i)
{
#pragma omp single
{
    for (i=0; i<N; i++) {
        double r = random_number();
        if (p_vec[i] > r) {
            #pragma omp task
            do_work (p_vec[i]);
        }
    }
}
}
Common mistakes

Race condition

```c
int nthreads;
#pragma omp parallel default(shared)
{
    nthreads = omp_get_num_threads();
}
```

Deadlock

```c
#pragma omp parallel
{
    ...
    #pragma omp critical
    {
        ...
        #pragma omp barrier
    }
}```
How about performance?

Factors that influence the performance of OpenMP programs:

- How the memory is accessed by individual threads
- The fraction of work that is sequential (or replicated)
- The overhead of handling OpenMP constructs
- Load imbalance
- Synchronization costs

Good programming practices:

- Optimize use of `barrier`
- Avoid `ordered` construct
- Avoid large `critical` blocks
- Maximize parallel regions
- Avoid parallel regions in inner loops
- Use `schedule(dynamic)` or `schedule(guided)` to address poor load balance
The issue of NUMA

- Non-uniform memory access (e.g., dual-socket quad-core Nehalem)

- Each thread should, if possible, only work with data close-by
  - Use of first touch in data initialization
  - Use of static scheduler with fixed chunk size
  - Avoid false sharing on ccNUMA architecture
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.