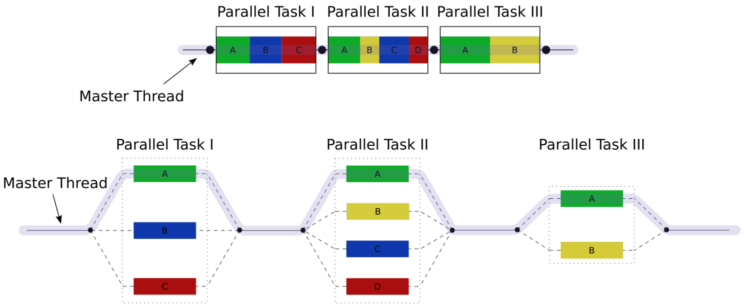
Parallel Programming with OpenMP

Introduction: OpenMP Programming Model

- Thread-based parallelism utilized on shared-memory platforms
- Parallelization is either explicit, where programmer has full control over parallelization or through using compiler directives, existing in the source code.
- Thread is a process of a code is being executed. A thread of execution is the smallest unit of processing.
- Multiple threads can exist within the same process and share resources such as memory

Introduction: OpenMP Programming Model

- Master thread is a single thread that runs sequentially; parallel execution occurs inside parallel regions and between two parallel regions, only the master thread executes the code. This is called the fork-join model:

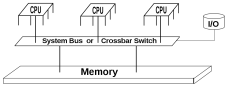


OpenMP Parallel Computing Hardware

Shared memory allows immediate access to all data from all processors without explicit communication.

Shared memory:

- multiple cpus are attached to the BUS
- all processors share the same primary memory
- the same memory address on different CPU's refer to the same memory location
- CPU-to-memory connection becomes a bottleneck: shared memory computers cannot scale very well



OpenMP versus MPI

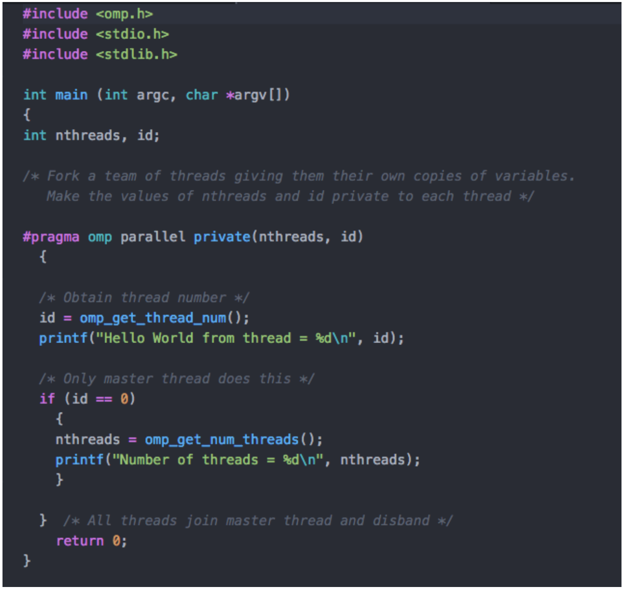
OpenMP (Open Multi-Processing):

- easy to use; loop-level parallelism
- non-loop-level parallelism is more difficult
- limited to shared memory computers
- cannot handle very large problems

An alternative is MPI (Message Passing Interface):

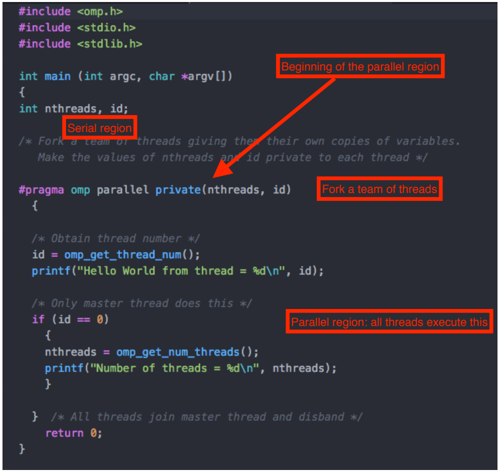
- require low-level programming; more difficult programming
- scalable cost/size
- can handle very large problems

OpenMP core structure



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

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

/* Fork a team of threads giving them their own copies of variables.
   Make the values of nthreads and id private to each thread */

#pragma omp parallel private(nthreads, id)
  {

  /* Obtain thread number */
  id = omp_get_thread_num();
  printf("Hello World from thread = %d\n", id);

  /* Only master thread does this */
  if (id == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  }  /* All threads join master thread and disband */
    return 0;
}
```

OpenMP core structure



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

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

/* Fork a team of threads giving them their own copies of variables.
   Make the values of nthreads and id private to each thread */

#pragma omp parallel private(nthreads, id)
  {

  /* Obtain thread number */
  id = omp_get_thread_num();
  printf("Hello World from thread = %d\n", id);

  /* Only master thread does this */
  if (id == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  }  /* All threads join master thread and disband */
    return 0;
}
```

Beginning of the parallel region
Serial region
Fork a team of threads
Parallel region: all threads execute this

Compiling:

export OMP_NUM_THREADS=4

gcc -o hello_omp -fopenmp hello_omp.c

OpenMP C Directives

A directive has a name followed by clauses

```
#include <omp.h>

int omp_get_num_threads(void)
\* in bash, export OMP_NUM_THREADS=8 *\

#pragma omp parallel default(shared) private(beta,pi)

#pragma omp parallel for
#pragma omp critical
#pragma omp single
#pragma omp barrier


#pragma omp parallel for reduction(+:sum)


double omp_get_wtime(void);
```

OpenMP C Directives: Reduction

A private copy for each list variable is created and initialized for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

Reduction

```
double avg, sum=0.0, A[MAX]; int i;
#pragma omp parallel for private ( sum )
          for (i = 0; i <= MAX ; i++)
                    sum += A[i];

avg = sum/MAX;
```

Flow dependence

```
double avg, sum=0.0, A[MAX]; int i;
#pragma omp for reduction(+ : sum)
          for (i = 0; i <= MAX ; i++)
                    sum += A[i];

avg = sum/MAX;
```

Dependence removed

```
 #pragma omp critical
            sum=sum+x
```