OpenMP : Multi-Processing API for Shared Memory

09/02/2020

Introduction to OpenMP

- Node-level parallel programming technique
  - Relies on shared memory
  - Builds on threads and vector primitives
  - User tells compiler how to parallelize code through pragma statements
- As with all shared memory programming, race conditions are a problem!
  - With pthreads, we saw *locks*
  - In OpenMP, we see *private variables*

Threading - OMP Parallel

- **#pragma omp parallel** initializes threads and executes all statements in following code block in parallel
  - Can be written as *#pragma omp parallel for* if followed by a single for loop… then the iterations of the loop are split across the threads
- **#pragma omp simd** performs vectorization on all code in block
  - Always followed by a for loop

Barriers

- Sometimes, all threads must wait at some point before moving on
- Think about dependencies in your program
  - May need to fully complete one function before moving onto the next
- **#pragma omp barrier** waits for all threads to reach this point before moving on
- There are also implicit barriers throughout OpenMP programs
- If you don’t want an implicit barrier **#pragma omp for nowait** : each thread continues executing immediately after for loop

Easiest Implementations

- **Loop level parallelism**: Idea, stick a #pragma omp statement before regular loops to parallelize them. Vectorize within parallel loops.

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {
#pragma omp simd
    for (int j = 0; j < N; j++)
        a[i][j] = b[i][j] + c[i][j]
} // Threads join barrier at parallel region completion
```

More difficult

- **Block level parallelization**: Create large region of code to be executed as independent thread.

```
#pragma omp parallel
{ // Everything in this region is parallel
#pragma omp for
    for (i = 0; i < N; i++) {
#pragma omp simd
        for (j = 0; j < N; j++)
            a[i][j] = b[i][j] + c[i][j]
    } // Do we need to barrier here or not?
#pragma omp for
    for (int i = 0; ...) {
        …
    }
}
```

Most Difficult

- **Program level parallelization**: want to use parallelism throughout as much of the program ass possible
  - When do you need to synchronize?
  - May have to re-think the entire program

Pragmas

- What is a pragma? Comment-like structure that tells the compiler additional information about the program
  - Processed when the program is compiled
- OMP Pragmas tell compiler:
  - How to parallelize program
  - Important information about program variables

```
C: #pragma omp directivename [optional clauses]
Fortran: !$OMP DirectiveName [optional clauses]
```

Examples

```
#pragma omp simd reduction(+:sum)
!$OMP PARALLEL NUM_THREADS(2) THREADPRIVATE(X)
```

Helpful Hints

- **omp_get_num_threads()** returns the number of threads executing the block
- **omp_get_thread_num()** returns the id of the calling thread
- **OMP_NUM_THREADS** environment variable sets the number of threads to be initialized for all loops which don’t specify num_threads
- -fopenmp compile flag for parallel code
- -fopenmp-simd compile flag for vectorized code

Environment Variables

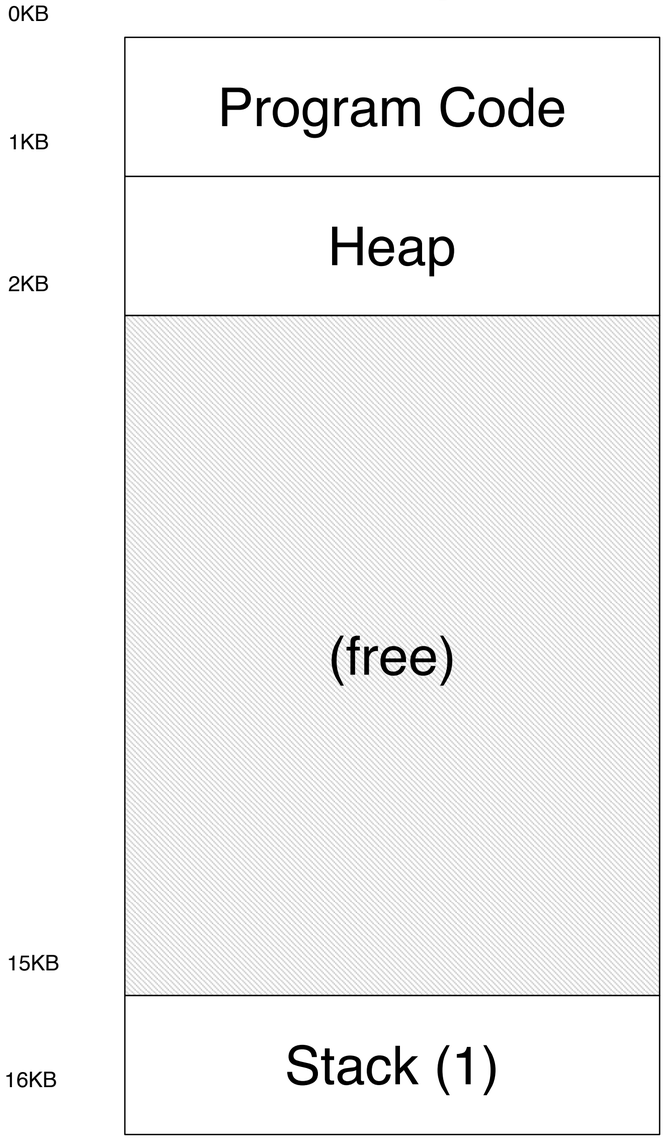
• OMP_DEFAULT_DEVICE:	Set the device used in target regions
• OMP_DYNAMIC:	Dynamic adjustment of threads
• OMP_MAX_ACTIVE_LEVELS :	Set the maximum number of nested parallel regions
• OMP_MAX_TASK_PRIORITY:	Set the maximum task priority value
• OMP_NESTED:	Nested parallel regions
• OMP_NUM_THREADS:	Specifies the number of threads to use
• OMP_PROC_BIND:	Whether theads may be moved between CPUs
• OMP_PLACES:	Specifies on which CPUs the threads should be placed
• OMP_STACKSIZE:	Set default thread stack size
• OMP_SCHEDULE:	How threads are scheduled
• OMP_THREAD_LIMIT:	Set the maximum number of threads
• OMP_WAIT_POLICY:	How waiting threads are handled

Race Conditions

- Threads : parallelism inside a single process
  - Mostly share the same address space
    - Heap : dynamically allocated global memory
    - Data Section : statically allocated global memory
  - A thread's local variables may be either ***shared*** or ***private***
  - **If multiple threads update one shared variable at the same time, you can have a race condition**

Race Conditions

A Single-Threaded Address Space

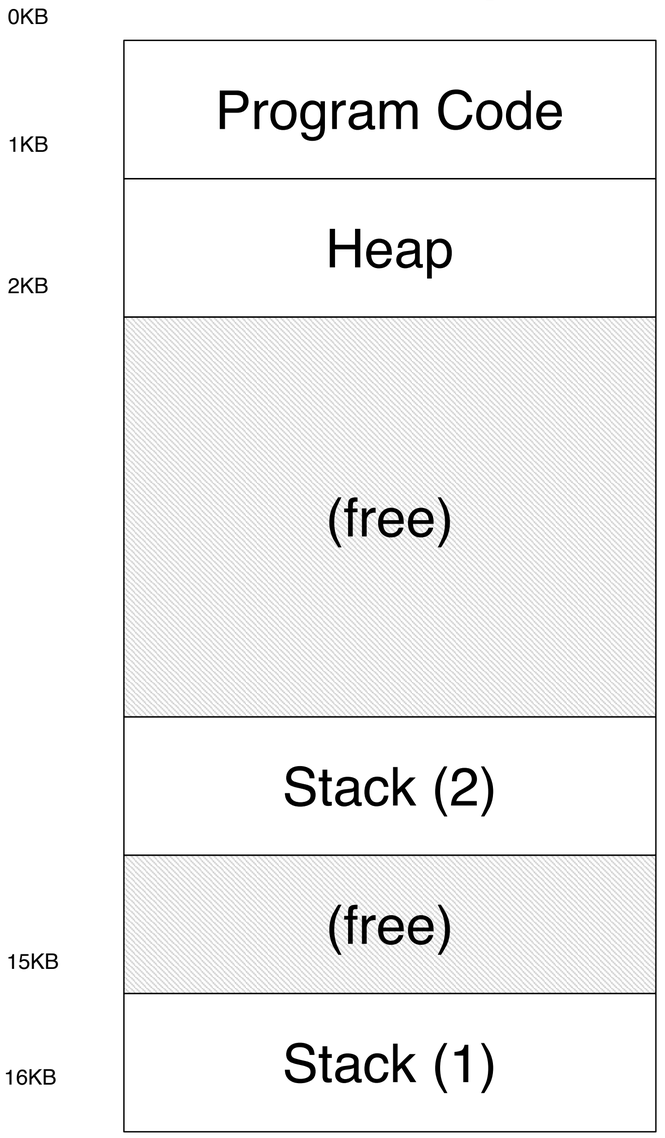


The code segment:
where instructions live

The heap segment:
contains malloc'd data
dynamic data structures
(it grows downward)

(it grows upward)
The stack segment:
contains local variables
arguments to routines,
return values, etc.

Two threaded Address Space



Race Conditions

- **OpenMP Example:**

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    int nthreads, thread_id;
#pragma omp parallel
    {
   nthreads = omp_get_num_threads();
   thread_id = omp_get_thread_num();
   printf("Goodbye slow serial world and Hello OpenMP!\n");
   printf(" I have %d thread(s) and my thread id is %d\n",
          nthreads,thread_id);
    }
}
```

Helpful Tips

- Multiple threads must not concurrently write to a shared variable
- **Solution 1:** Get rid of the shared variable
  - In HelloWorld case, each thread could have its own **int thread_id**
- **Solution 2:** Break code into independent sections
  - This happens in terms of the reduction clause

How do I parallelize this?

```
int i, j;
double sum = 0, diff = 0;
for (int i = 1; i < (N – 1); i++)
    for (int j = 1; j < (N – 1); j++) {
   sum = old[i-1][j] + old[i+1][j] + old[i][j-1] + old[i][j+1];
   new[i][j] = sum/4.0;
   diff = MAX(diff, new[i][j] – old[i][j]);
    }
```