CSC510
Parallel Programming

Shared Memory Programming
(Part 2)

Outline

- OpenMP
- Shared-memory model
- Parallel for loops
- Declaring private variables
- Critical sections
- Reductions
- Performance improvements
- More general data parallelism
- Functional parallelism
More General Data Parallelism

- Our focus has been on the parallelization of `for` loops
- Other opportunities for data parallelism
  - processing items on a “to do” list
  - `for` loop + additional code outside of loop

Processing a “To Do” List
int main (int argc, char *argv[]) {
    struct job_struct *job_ptr;
    struct task_struct *task_ptr;

    ...
    task_ptr = get_next_task (&job_ptr);
    while (task_ptr != NULL) {
        complete_task (task_ptr);
        task_ptr = get_next_task (&job_ptr);
    }
    ...
}

char *get_next_task (struct job_struct **job_ptr) {
    struct task_struct *answer;

    if (*job_ptr == NULL) answer = NULL;
    else {
        answer = (*job_ptr)->task;
        *job_ptr = (*job_ptr)->next;
    }
    return answer;
}
Parallelization Strategy

- Every thread should repeatedly take next task from list and complete it, until there are no more tasks
- We must ensure no two threads take same task from the list; i.e., must declare a critical section

parallel Pragma

- The parallel pragma precedes a block of code that should be executed by all of the threads
- Note: execution is replicated among all threads

```c
#pragma omp parallel private(task_ptr)
{
    task_ptr = get_next_task (&job_ptr);
    while (task_ptr != NULL) {
        complete_task (task_ptr);
        task_ptr = get_next_task (&job_ptr);
    }
}
```

How to ensure function `get_next_task` executes atomically?
Critical Section for \texttt{get\_next\_task}

\begin{verbatim}
char *get_next_task(struct job_struct **job_ptr) {
    struct task_struct *answer;
    #pragma omp critical
    {
        if (*job_ptr == NULL) answer = NULL;
        else {
            answer = (*job_ptr)->task;
            *job_ptr = (*job_ptr)->next;
        }
    }
    return answer;
}
\end{verbatim}

PP-4: Shared Memory Programming

Functions for SPMD-style Programming

\begin{itemize}
  \item The parallel pragma allows us to write SPMD-style programs
  \item In these programs we often need to know number of threads and thread ID number
  \item OpenMP provides functions to retrieve this information
\end{itemize}
**Function omp_get_thread_num**

- This function returns the thread identification number
- If there are \( t \) threads, the ID numbers range from 0 to \( t-1 \)
- The master thread has ID number 0

```c
int omp_get_thread_num (void)
```

**Function omp_get_num_threads**

- Function `omp_get_num_threads` returns the number of active threads
- If call this function from sequential portion of program, it will return 1

```c
int omp_get_num_threads (void)
```
for Pragma

- The **parallel** pragma instructs every thread to execute all of the code inside the block
- If we encounter a **for** loop that we want to divide among threads, we use the **for** pragma

```c
#pragma omp for
```

Example

```c
for (i = 0; i < m; i++) {
    low = a[i];
    high = b[i];
    if (low > high) {
        printf ("Exiting (%d)\n", i);
        break;
    }
    for (j = low; j < high; j++)
        c[j] = (c[j] - a[i])/b[i];
}
```

Can we execute the outer loop in parallel?
Can we put a parallel for pragma before the loop indexed by j?
Can we invert for loop?
Can we put parallel pragma in front of the loop indexed by i?
Example Use of for Pragma

```c
#pragma omp parallel private(i,j)
for (i = 0; i < m; i++) {
    low = a[i];
    high = b[i];
    if (low > high) {
        printf("Exiting (%d)\n", i);
        break;
    }
#pragma omp for
    for (j = low; j < high; j++)
        c[j] = (c[j] - a[i])/b[i];
}
```

single Pragma

- Suppose we only want to see the output once
- The **single** pragma directs compiler that only a single thread should execute the block of code the pragma precedes
- Syntax:

```c
#pragma omp single
```
Use of single Pragma

```c
#pragma omp parallel private(i,j)
for (i = 0; i < m; i++) {
    low = a[i];
    high = b[i];
    if (low > high) {
        #pragma omp single
        printf("Exiting (%d)\n", i);
        break;
    }
    #pragma omp for
    for (j = low; j < high; j++)
        c[j] = (c[j] - a[i])/b[i];
}
```

nowait Clause

- Compiler puts a barrier synchronization at end of every parallel for statement
- In our example, this is necessary: if a thread leaves loop and changes `low` or `high`, it may affect behavior of another thread
- If we make these private variables, then it would be okay to let threads move ahead, which could reduce execution time
Use of `nowait` Clause

```c
#pragma omp parallel private(i,j,low,high)
for (i = 0; i < m; i++) {
    low = a[i];
    high = b[i];
    if (low > high) {
        #pragma omp single
        printf("Exiting (%d)\n", i);
        break;
    }
    #pragma omp for nowait
    for (j = low; j < high; j++)
        c[j] = (c[j] - a[i])/b[i];
}
```

Functional Parallelism

- To this point all of our focus has been on exploiting data parallelism
- OpenMP allows us to assign different threads to different portions of code (functional parallelism)
Functional Parallelism Example

\[
v = \text{alpha}();
\]
\[
w = \text{beta}();
\]
\[
x = \text{gamma}(v, w);
\]
\[
y = \text{delta}();
\]
\[
\text{printf} \left( \"%6.2f\n\", \text{epsilon}(x,y) \right);
\]

May execute alpha, beta, and delta in parallel

parallel sections Pragma

- Precedes a block of \( k \) blocks of code that may be executed concurrently by \( k \) threads
- Syntax:

\[
\text{#pragma omp parallel sections}
\]
**section Pragma**

- Precedes each block of code within the encompassing block preceded by the parallel sections pragma
- May be omitted for first parallel section after the parallel sections pragma
- Syntax:
  ```
  #pragma omp section
  ```

**Example of parallel sections**

```c
#pragma omp parallel sections
{
  #pragma omp section /* Optional */
  v = alpha();
  #pragma omp section
  w = beta();
  #pragma omp section
  y = delta();
}
  x = gamma(v, w);
  printf("%6.2f\n", epsilon(x,y));
```
Another Approach

Execute alpha and beta in parallel. Execute gamma and delta in parallel.

sectionsPragma

- Appears inside a parallel block of code
- Has same meaning as the \texttt{parallel sections} pragma
- If multiple \texttt{sections} pragmas inside one parallel block, may reduce fork/join costs
Use of sections Pragma

```c
#pragma omp parallel
{
    #pragma omp sections
    {
        v = alpha();
        #pragma omp section
        w = beta();
    }
    #pragma omp sections
    {
        x = gamma(v, w);
        #pragma omp section
        y = delta();
    }
}
printf("%.2f\n", epsilon(x, y));
```

Summary (1/3)

- **OpenMP** an API for shared-memory parallel programming
- Shared-memory model based on fork/join parallelism
- Data parallelism
  - parallel for pragma
  - reduction clause
Summary (2/3)

- Functional parallelism (parallel sections pragma)
- SPMD-style programming (parallel pragma)
- Critical sections (critical pragma)
- Enhancing performance of parallel for loops
  - Inverting loops
  - Conditionally parallelizing loops
  - Changing loop scheduling

Summary (3/3)

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>OpenMP</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suitable for multiprocessors</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Suitable for multicomputers</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Supports incremental parallelization</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Minimal extra code</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Explicit control of memory hierarchy</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
CSC510
Parallel Programming

Questions?

Dr. Joe Zhang
PP_4: Shared-Memory Programming