Introduction to OpenMP

www.openmp.org

Slides adopted from CS236370
Motivation

• Parallelize the following code using threads:
  
  ```
  for (i=0; i<n; i++) {
      sum = sum+sqrt(sin(data[i]));
  }
  ```

• A lot of work to do a simple thing

• Different threading APIs:
  
  - Windows: CreateThread
  - UNIX: pthread_create

• Problems with the code:
  
  - Need mutex to protect the accesses to sum
  - Different code for serial and parallel version
  - No built-in tuning (# of processors someone?)
Motivation: OpenMP

- A language extension that introduces parallelization constructs into the language
- Parallelization is orthogonal to the functionality
  - If the compiler does not recognize the OpenMP directives, the code remains functional (albeit single-threaded)
- Based on shared-memory multithreaded programming
- Includes constructs for parallel programming: critical sections, atomic access, variable privatization, barriers etc.
- Industry standard
  - Supported by Intel, Microsoft, Sun, IBM, HP, etc.
    Some behavior is implementation-dependent
  - Intel compiler available for Windows and Linux
OpenMP execution model

Fork and Join: Master thread spawns a team of threads as needed
OpenMP basics

• Most constructs in OpenMP* are compiler directives or pragmas.
  • For C and C++, the pragmas take the form:

    \#pragma omp construct [clause [clause]…]

• Main construct:
  \#pragma omp parallel
  – Defines a parallel region over structured block of code
  – Threads are created as ‘parallel’ pragma is crossed
  – Threads block at end of region
double D[1000];
#pragma omp parallel
{
    int i; double sum = 0;
    for (i=0; i<1000; i++) sum += D[i];
    printf("Thread %d computes %f\n", omp_thread_num(), sum);
}

- Executes the same code several times (as many as there are threads)
  - How many threads do we have?
    omp_set_num_threads(n). Or, environment variable OMP_NUM_THREADS
  - What is the use of repeating the same work several times in parallel?
    Can use omp_thread_num() to distribute the work between threads.

- D is shared between the threads, i and sum are private
• Usage:
  • OpenMP switches: `-openmp` : `/Qopenmp`
  • OpenMP reports: `-openmp-report` : `/Qopenmp-report`

```
gcc -fopenmp prog.c -o prog
OMP_NUM_THREADS=2; prog
```
• Each \texttt{#pragma omp parallel} creates a team of threads, which exist as long as the following block executes
  
  – \texttt{#pragma omp for} and \texttt{#pragma omp section} must be placed within a \texttt{#pragma omp parallel}.
  
  – Optimization: If there are several \texttt{#pragma omp for} and/or \texttt{#pragma omp section} within the same parallel, the threads will not be destroyed and created again
Work-sharing: the for loop

- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct
- (nowait can be used to prevent waiting)
answer1 = long_computation_1();
answer2 = long_computation_2();
if (answer1 != answer2) { ... }

• How to parallelize?
  - These are just two independent computations!

#pragma omp sections
{
  #pragma omp section
  answer1 = long_computation_1();
  #pragma omp section
  answer2 = long_computation_2();
}
if (answer1 != answer2) { ... }
Comparison

Sequential code

\[
\text{for (int } i=0; i<N; i++) \{ \text{a}[i]=b[i]+c[i]; } \]

(Semi) manual parallelization

\[
\#pragma \text{omp parallel} \\
\{ \\
\text{int id = omp_get_thread_num();} \\
\text{int Nthr = omp_get_num_threads();} \\
\text{int istart = id*}N\text{/Nthr, iend = (id+1)*}N\text{/Nthr;} \\
\text{for (int } i=\text{istart;} i<i\text{end;} i++) \{ \text{a}[i]=b[i]+c[i]; } \}
\]

Automatic parallelization of the for loop

\[
\#pragma \text{omp parallel} \\
\#pragma \text{omp for schedule(static)} \\
\{ \\
\text{for (int } i=0; i<N; i++) \{ \text{a}[i]=b[i]+c[i]; } \}
\]
Notes on \texttt{#parallel for}

- Only simple kinds of for loops are supported
  - One signed integer variable in the loop.
  - Initialization: \texttt{var=init}
  - Comparison: \texttt{var op last}, \texttt{op: <, >, <=, >=}
  - Increment: \texttt{var++}, \texttt{var--}, \texttt{var+=incr}, \texttt{var-=incr}, etc.
  - All of \texttt{init}, \texttt{last}, \texttt{incr} must be loop invariant

- Can combine the parallel and work sharing directives: \texttt{#pragma omp parallel for} …
Optimizations of `parallel for`

- **Load balancing**
  - If some iterations are faster, some processors may get **idle**
  - We don’t always know the distribution of work, may need to re-distribute **dynamically**

- **Granularity**
  - Thread creation and synchronization are costly
  - Assigning work to threads per-iteration is costly

- **Trade-off between load balancing and granularity!**
Assigning iterations

The `schedule` clause affects how loop iterations are mapped onto threads

**schedule(static [,chunk])**
- Blocks of iterations of size “chunk” to threads
- Round robin distribution

**schedule(dynamic [,chunk])**
- Threads grab “chunk” iterations
- When done with iterations, thread requests next set
- Chunk=1 by default

**schedule(guided [,chunk])**
- Dynamic schedule starting with large block
- Size of the blocks shrink; no smaller than “chunk”

<table>
<thead>
<tr>
<th>When to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predictable and similar work per iteration</td>
</tr>
<tr>
<td><strong>Small iteration size</strong></td>
</tr>
<tr>
<td>Unpredictable, highly variable work per iteration</td>
</tr>
<tr>
<td><strong>Large iteration size</strong></td>
</tr>
<tr>
<td>Special case of dynamic to reduce scheduling overhead</td>
</tr>
</tbody>
</table>
Example: What schedule to use?

- The function TestForPrime (usually) takes little time
  - But can take long, if the number is a prime indeed

```c
#pragma omp parallel for schedule ????
for( int i = start; i <= end; i += 2 )
{
    if ( TestForPrime(i) ) gPrimesFound++;
}
```
Controlling Granularity

- `#pragma omp parallel if (expression)`
  - Can be used to disable parallelization in some cases (e.g. when the input is determined to be too small)

- `#pragma omp num_threads (expression)`
  - Control the number of threads used for this parallel region
OpenMP: Data Environment

• Shared Memory programming model
  – Most variables (including locals) are shared by default – unlike Pthreads!
    
    ```c
    { 
      int sum = 0;
      #pragma omp parallel for
      for (int i=0; i<N; i++) sum += i;
    }
    ```
  – Global variables are shared

• Some variables can be private
  – Automatic variables inside the statement block
  – Automatic variables in the called functions
  – Variables can be explicitly declared as private. In that case, a local copy is created for each thread
Overriding storage attributes

- **private:**
  - A copy of the variable is created for each thread
  - There is no connection between the original variable and the private copies
  - Can achieve the same using variables inside `{ }`

- **firstprivate:**
  - Same, but the initial value of the variable is copied from the main copy

- **lastprivate:**
  - Same, but the last value of the variable is copied to the main copy

```c
int i;
#pragma omp parallel for private(i)
for (i=0; i<n; i++) { … }
```

```c
int idx=1;
int x = 10;
#pragma omp parallel for 
  firstprivate(x) lastprivate(idx)
for (i=0; i<n; i++) {
  if (data[i]==x) idx = i;
}
```
Reduction

for (j=0; j<N; j++) {
    sum = sum+a[j]*b[j];
}

• How to parallelize this code?
  - sum is not private, but accessing it atomically is too expensive
  - Have a private copy of sum in each thread, then add them up

• Use the reduction clause!
  #pragma omp parallel for reduction(+: sum)
  - Any associative operator must be used: +, -, ||, |, *, etc.
  - The private value is initialized automatically (to 0, 1, ~0 …)
#pragma omp reduction

- Syntax: `#pragma omp reduction (op:list)`
- The variables in “list” must be shared in the enclosing parallel region
- Inside parallel or work-sharing construct:
  - A PRIVATE copy of each list variable is created and initialized depending on the “op”
  - These copies are updated locally by threads
  - At end of construct, local copies are combined through “op” into a single value and combined with the value in the original SHARED variable

```c
float dot_prod(float* a, float* b, int N) {
    float sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for(int i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```
OpenMP memory model

• Shared memory model
  - Threads communicate by accessing shared variables

• The sharing is defined syntactically
  - Any variable that is seen by two or more threads is shared
  - Any variable that is seen by one thread only is private

• Race conditions possible
  - Use synchronization to protect from conflicts
  - Change how data is stored to minimize the synchronization
X = 0;
#pragma omp parallel
X = X+1;
X = X+1;

• What should the result be (assuming 2 threads)?
  - 2 is the expected answer
  - But can be 1 with unfortunate interleaving

• OpenMP assumes that the programmer knows what (s)he is doing
  - Regions of code that are marked to run in parallel are independent
  - If access collisions are possible, it is the programmer’s responsibility to insert protection
Many of the existing mechanisms for shared programming
- Single/Master execution
- Critical sections, Atomic updates
- Ordered
- Barriers
- Nowait (turn synchronization off!)
- Flush (memory subsystem synchronization)
- Reduction (already seen)
### Single/Master

- **#pragma omp single**
  - Only one of the threads will execute the following block of code
  - The rest will wait for it to complete
  - Good for non-thread-safe regions of code (such as I/O)
  - Must be used in a parallel region
  - Applicable to `parallel` for sections

- **#pragma omp master**
  - The following block of code will be executed by the master thread
  - No synchronization involved
  - Applicable only to `parallel` sections

---

#### Example:

```c
#pragma omp parallel
{
  do_preprocessing();
  #pragma omp single
  read_input();
  #pragma omp master
  notify_input_consumed();
  do_processing();
}
```
Critical Sections

- **#pragma omp critical [name]**
  - Standard critical section functionality

- Critical sections are global in the program
  - Can be used to protect a single resource in different functions

- Critical sections are identified by the name
  - All the unnamed critical sections are mutually exclusive throughout the program
  - All the critical sections having the same name are mutually exclusive between themselves
• Notify the compiler that the variable is shared

```c
float dot_prod(float* a, float* b, int N) {
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

• What’s the problem here?
• Protect shared variables from data races

```c
float dot_prod(float* a, float* b, int N) {
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
        for(int i=0; i<N; i++) {
            #pragma omp critical
                sum += a[i] * b[i];
        }
    return sum;
}
```

• Another option: use `#pragma omp atomic`
  - Can protect only a single assignment
  - Generally faster than critical
Atomic execution

• Critical sections on the cheap
  – Protects a single variable update
  – Can be much more efficient (a dedicated assembly instruction on some architectures)

• `#pragma omp atomic update_statement`

• Update statement is one of: `var= var op expr, var op= expr, var++, var--`.  
  – The variable must be a scalar  
  – The operation op is one of: `+, -, *, /, ^, &, |, <<, >>`  
  – The evaluation of expr is not atomic!
#pragma omp ordered statement
  - Executes the statement in the sequential order of iterations

Example:

```c
#pragma omp parallel for
for (j=0; j<N; j++) {
  int result = heavy_computation(j);
  #pragma omp ordered
  printf("computation(%d) = %d\n", j, result);
}
```
Barrier synchronization

- `#pragma omp barrier`
- Performs a barrier synchronization between all the threads in a team *at the given point*.
- Example:

```c
#pragma omp parallel
{
    int result = heavy_computation_part1();
    #pragma omp atomic
    sum += result;
    #pragma omp barrier
    heavy_computation_part2(sum);
}
```
Controlling OpenMP behavior

- **omp_set_dynamic(int)/omp_get_dynamic()**
  - Allows the implementation to adjust the number of threads dynamically

- **omp_set_num_threads(int)/omp_get_num_threads()**
  - Control the number of threads used for parallelization (maximum in case of dynamic adjustment)
  - Must be called from sequential code
  - Also can be set by OMP_NUM_THREADS environment variable

- **omp_get_numprocs()**
  - How many processors are currently available?

- **omp_get_thread_num()**

- **omp_set_nested(int)/omp_get_nested()**
  - Enable nested parallelism

- **omp_in_parallel()**
  - Am I currently running in parallel mode?

- **omp_get_wtime()**
  - A portable way to compute wall clock time
Performance Issues

- Idle threads do no useful work
- Divide work among threads as evenly as possible
  - Threads should finish parallel tasks at same time
- Synchronization may be necessary
  - Minimize time waiting for protected resources
- Parallelization Granularity may be too low
Load Imbalance

- Unequal work loads lead to idle threads and wasted time.
  - Need to distribute the work as evenly as possible!

```c
#pragma omp parallel
{
  #pragma omp for
  for( ; ; ){
    
  }
}
```
Synchronization

- Lost time waiting for locks
  - Prefer to use structures that are as lock-free as possible!
  - Use parallelization granularity which is as large as possible.

```c
#pragma omp parallel
{
  #pragma omp critical
  {
    ...
  }
  ...
}
```
f(x) = \frac{4.0}{1+x^2}

Example: Parallel Numerical Integration

```c
static long num_steps=100000;
double step, pi;

void main()
{
    int i;
    double x, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```
Computing Pi through integration

Parallelize the numerical integration code using OpenMP

What variables can be shared?

What variables need to be private?

What variables should be set up for reductions?

```c
static long num_steps=100000;
double step, pi;

void main()
{
    int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```
Computing Pi through integration

```c
static long num_steps=100000;
double step, pi;

void main()
{
    int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    #pragma omp parallel for \ 
        private(x) reduction(+:sum)
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

i is private since it is the loop variable