Outline

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

- An application programming interface (API) for shared-memory parallel programming.
  - Compiler directives
  - Library of support functions
- MP = multiprocessing
- Designed for systems in which each thread or process can potentially have access to all available memory.
- System is viewed as a collection of cores or CPU's, all of which have access to main memory.
- OpenMP works in conjunction with Fortran, C, or C++
- Visit the OpenMP website at http://www.openmp.org

Shared-memory Model

Processors interact and synchronize with each other through shared variables.
**Fork/Join Parallelism**

- Initially only master thread is active
- Master thread executes sequential code
- Fork: Master thread creates or awakens additional threads to execute parallel code
- Join: At end of parallel code created threads die or are suspended

---

**Shared-memory Model vs. Message-passing Model (#1)**

- Shared-memory model
  - Number active threads 1 at start and finish of program, changes dynamically during execution
- Message-passing model
  - All processes active throughout execution of program
Incremental Parallelization

- Sequential program a special case of a shared-memory parallel program
- Parallel shared-memory programs may only have a single parallel loop
- Incremental parallelization: process of converting a sequential program to a parallel program a little bit at a time

Shared-memory Model vs. Message-passing Model (#2)

- Shared-memory model
  - Execute and profile sequential program
  - Incrementally make it parallel
  - Stop when further effort not warranted
- Message-passing model
  - Sequential-to-parallel transformation requires major effort
  - Transformation done in one giant step rather than many tiny steps
Parallel for Loops

- C programs often express data-parallel operations as `for` loops
  
  ```c
  for (i = first; i < size; i += prime)
      marked[i] = 1;
  ```

- OpenMP makes it easy to indicate when the iterations of a loop may execute in parallel
- Compiler takes care of generating code that forks/joins threads and allocates the iterations to threads

Pragmas

- Pragma: a compiler directive in C or C++
- Stands for “pragmatic information”
- A way for the programmer to communicate with the compiler
- Compiler free to ignore pragmas
- Syntax:
  ```c
  #pragma omp <rest of pragma>
  ```
**Parallel for Pragma**

- Format:
  
  ```
  #pragma omp parallel for
  for (i = 0; i < N; i++)
      a[i] = b[i] + c[i];
  ```

- Compiler must be able to verify the run-time system will have information it needs to schedule loop iterations

**Shared and Private Variables**

- Shared variable: has same address in execution context of every thread
- Private variable: has different address in execution context of every thread
- A thread cannot access the private variables of another thread
int main (int argc, char *argv[]) {
    int b[3];
    char *cptr;
    int i;
    cptr = malloc(1);
    #pragma omp parallel for
    for (i = 0; i < 3; i++)
        b[i] = i;
}

index i is a private variable and each thread has its own copy.
b and cptr are shared variables

Function omp_get_num_procs

• Returns number of physical processors available for use by the parallel program

int omp_get_num_procs (void)
Function omp_set_num_threads

- Uses the parameter value to set the number of threads to be active in parallel sections of code
- May be called at multiple points in a program

```c
void omp_set_num_threads (int t)
```

Pop Quiz:
Write a C program segment that sets the number of threads equal to the number of processors that are available.

Declaring Private Variables

```c
for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
    for (j = 0; j < n; j++)
        a[i][j] = MIN(a[i][j],a[i][k]+tmp);
```

- Either loop could be executed in parallel
- We prefer to make outer loop parallel, to reduce number of forks/joins
- We then must give each thread its own private copy of variable j
### private Clause

- Clause: an optional, additional component to a pragma
- Private clause: directs compiler to make one or more variables private

```plaintext
private ( <variable list> )
```

### Example Use of private Clause

```plaintext
#pragma omp parallel for private(j)
for (i = 0; i < BLOCK_SIZE(id, p, n); i++)
  for (j = 0; j < n; j++)
    a[i][j] = MIN(a[i][j], a[i][k]+tmp);
```
firstprivate Clause

• Used to create private variables having initial values identical to the variable controlled by the master thread as the loop is entered
• Variables are initialized once per thread, not once per loop iteration
• If a thread modifies a variable’s value in an iteration, subsequent iterations will get the modified value

lastprivate Clause

• Sequentially last iteration: iteration that occurs last when the loop is executed sequentially
• lastprivate clause: used to copy back to the master thread’s copy of a variable the private copy of the variable from the thread that executed the sequentially last iteration
Critical Sections

Consider this C program segment to compute $\pi$ using the rectangle rule:

```c
double area, pi, x;
int i, n;
...
area = 0.0;
for (i = 0; i < n; i++) {
    x += (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Race Condition

- If we simply parallelize the loop...

```c
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```
Race Condition (cont.)

- ... we set up a race condition in which one process may “race ahead” of another and not see its change to shared variable \texttt{area}

\[
\text{area } \begin{array}{c} 15.230 \end{array} \quad \text{Answer should be 18.995}
\]

Thread A \begin{array}{c} 15.432 \end{array} \quad \text{Thread B } \begin{array}{c} 15.230 \end{array}

\[\text{area } += \frac{4.0}{1.0 + x \times x}\]

Race Condition Time Line

<table>
<thead>
<tr>
<th>Value of area</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.667</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.667</td>
<td>+ 3.765</td>
<td></td>
</tr>
<tr>
<td>15.432</td>
<td></td>
<td>+ 3.563</td>
</tr>
<tr>
<td>15.230</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Critical Pragma**

- Critical section: a portion of code that only thread at a time may execute
- We denote a critical section by putting the pragma

```c
#pragma omp critical
```

in front of a block of C code

**Correct, But Inefficient, Code**

```c
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
#pragma omp critical
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```
Source of Inefficiency

- Update to *area* inside a critical section
- Only one thread at a time may execute the statement; i.e., it is sequential code
- Time to execute statement significant part of loop
- By Amdahl’s Law we know speedup will be severely constrained

Reductions

- Reductions are so common that OpenMP provides support for them
- May add reduction clause to `parallel for` pragma
- Specify reduction operation and reduction variable
- OpenMP takes care of storing partial results in private variables and combining partial results after the loop
reduction Clause

- The reduction clause has this syntax:
  
  reduction (<op> :<variable>)

- Operators
  - + Sum
  - * Product
  - & Bitwise and
  - | Bitwise or
  - ^ Bitwise exclusive or
  - && Logical and
  - || Logical or

π-finding Code with Reduction Clause

double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for \n  private(x) reduction(+:area)
for (i = 0; i < n; i++) {
  x = (i + 0.5)/n;
  area += 4.0/(1.0 + x*x);
}
pi = area / n;
Performance with Reduction

Execution times on a Sun Enterprise Server 4000 of two programs that use
the rectangle rule.

<table>
<thead>
<tr>
<th>Using critical pragma</th>
<th>Using reduction clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0780</td>
<td>0.0273</td>
</tr>
<tr>
<td>0.1510</td>
<td>0.0146</td>
</tr>
<tr>
<td>0.3400</td>
<td>0.0105</td>
</tr>
<tr>
<td>0.3608</td>
<td>0.0086</td>
</tr>
<tr>
<td>0.4710</td>
<td>0.0076</td>
</tr>
</tbody>
</table>

When n=100,000, speedup is 3.16

Performance Improvement #1

- Too many fork/joins can lower performance
- Inverting loops may help performance if
  - Parallelism is in inner loop
  - After inversion, the outer loop can be made parallel
  - Inversion does not significantly lower cache hit rate
Performance Improvement #2

- If loop has too few iterations, fork/join overhead is greater than time savings from parallel execution.
- The `if` clause instructs compiler to insert code that determines at run-time whether loop should be executed in parallel; e.g.,

  ```
  #pragma omp parallel for if(n > 5000)
  ```

<table>
<thead>
<tr>
<th>Threads</th>
<th>n = 100</th>
<th>n = 100,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.964</td>
<td>27.288</td>
</tr>
<tr>
<td>2</td>
<td>1.436</td>
<td>14.598</td>
</tr>
<tr>
<td>3</td>
<td>1.732</td>
<td>10.506</td>
</tr>
<tr>
<td>4</td>
<td>1.990</td>
<td>8.648</td>
</tr>
</tbody>
</table>

**Table 17.3** Execution time on a Sun Enterprise Server 4000 of a parallel C program that computes \( \pi \) using the rectangle rule, as a function of number of rectangles and number of threads.

Performance Improvement #3

- We can use `schedule` clause to specify how iterations of a loop should be allocated to threads.
- Static schedule: all iterations allocated to threads before any iterations executed.
- Dynamic schedule: only some iterations allocated to threads at beginning of loop's execution. Remaining iterations allocated to threads that complete their assigned iterations.
Static vs. Dynamic Scheduling

- **Static scheduling**
  - Low overhead
  - May exhibit high workload imbalance

- **Dynamic scheduling**
  - Higher overhead
  - Can reduce workload imbalance

Chunks

- A chunk is a contiguous range of iterations
- Increasing chunk size reduces overhead and may increase cache hit rate
- Decreasing chunk size allows finer balancing of workloads
**schedule Clause**

- Syntax of schedule clause
  
  \[
  \text{schedule (}<\text{type}>, [<\text{chunk}>])
  \]

- Schedule type required, chunk size optional

- Allowable schedule types
  - static: static allocation
  - dynamic: dynamic allocation
  - guided: guided self-scheduling
  - runtime: type chosen at run-time based on value of environment variable OMP_SCHEDULE

**Scheduling Options**

- schedule(static): block allocation of about n/t contiguous iterations to each thread
- schedule(static,C): interleaved allocation of chunks of size C to threads
- schedule(dynamic): dynamic one-at-a-time allocation of iterations to threads
- schedule(dynamic,C): dynamic allocation of C iterations at a time to threads
Scheduling Options (cont.)

• schedule(guided, C): dynamic allocation of chunks to tasks using guided self-scheduling heuristic. Initial chunks are bigger, later chunks are smaller, minimum chunk size is C.

• schedule(guided): guided self-scheduling with minimum chunk size 1

• schedule(runtime): schedule chosen at run-time based on value of OMP_SCHEDULE; Unix example:
  
  ```
  setenv OMP_SCHEDULE "static,1"
  ```