Lecture 4

Multithreading
OpenMP
Parallel Performance
Announcements
Today’s Lecture

- Multithreading
- OpenMP
- Parallel Performance Metrics
Assignment #1

• Blocking for cache will boost performance but a lot more is needed to approach ATLAS’ performance.

\[ R_{\infty} = 4 \times 2.33 = 9.32 \text{ Gflops} \]

\[ \approx 87\% \text{ of peak} \]
Threads programming model

- Extends traditional von Neumann execution model to run multiple instructions stream in parallel
- Program executes a collection of independent instruction streams, called *threads*
- A thread is similar to a procedure call with notable differences
  - A new storage class: shared data
  - A procedure call is “synchronous:” a return indicates completion
  - A spawned thread executes asynchronously until it completes
  - Both share global storage with caller
  - Synchronization may be needed when updating shared state (thread safety)
- Usually implemented under SPMD programming model

<table>
<thead>
<tr>
<th>Private</th>
<th>Shared</th>
</tr>
</thead>
</table>

**Same Program Multiple Data**
- Each thread is initialized with the same code
- Threads are distinguished by their rank
- We will generally specify number of threads at program’s start
Why threads?

- Processes are “heavy weight” objects scheduled by the OS
  - Protected address space, open files, and other state

- A thread AKA a lightweight process (LWP)
  - Threads share the address space and open files of the parent, but have their own stack
  - Reduced management overheads, e.g. thread creation
  - Kernel scheduler multiplexes threads
Threads Programming model

- Start with a single root thread
- Fork-join parallelism to create concurrently executing threads
- Threads communicate via shared memory
- A spawned thread executes asynchronously until it completes
- Threads may or may not execute on different processors
Multithreading in Practice

• C++11

• POSIX Threads “standard” (pthreads):
  IEEE POSIX 1003.1c-1995
  ♦ Low level interface
  ♦ Beware of non-standard features

• Java threads not used in high performance computation

• OpenMP – program annotations

• Parallel programming languages
  ♦ Co-array FORTRAN
  ♦ UPC
C++11 Threads

• Via `<thread>`, C++ supports a threading interface similar to pthreads, though a bit more user friendly
• Async is a higher level interface suitable for certain kinds of applications
• New memory model
• Atomic template
Hello world with <Threads>

```
#include <thread>

void Hello(int TID) {
    cout << "Hello from thread " << TID << endl;
}

int main(int argc, char *argv[ ]) {
    thread *thrds = new thread[NT];

    // Spawn threads
    for(int t=0; t<NT; t++){
        thrds[t] = thread(Hello, t);
    }

    // Join threads
    for(int t=0; t<NT; t++)
        thrds[t].join();
}
```

$ ./hello_th 3
Hello from thread 0
Hello from thread 1
Hello from thread 2

$ ./hello_th 4
Running with 4 threads
Hello from thread 0
Hello from thread 3
Hello from thread 2
Hello from thread 1
Today’s Lecture

• Multithreading
• OpenMP
• Parallel Performance Metrics
OpenMP

• A higher level interface for threads programming
• Parallelization handled via source code annotations
• See http://www.openmp.org
• Compare with explicit threads programing
• Set the number of threads via the environment
  export OMP_NUM_THREADS=4

```c
#pragma omp parallel private(i) shared(n)
{
    #pragma omp for
    for(i=0; i < n; i++)
        work(i);
}
```
OpenMP’s Fork-Join Model

- A program begins life as a single thread
- Parallel regions spawn work groups of multiple threads
- The lexically enclosed program statements execute in parallel by all team members
- When we reach the end of the scope…
  - The team of threads synchronize at a barrier and are disbanded; they enter a wait state
  - Only the initial thread continues
- Thread teams can be created and disbanded many times during program execution, but this can be costly
- A clever compiler can avoid so many thread creations and joins
Fork join model with loops

```cpp
cout << "Serial\n";
N = 1000;
#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = B[i] + C[i];

M = 500;
#pragma omp parallel for
for (j=0; j<M; j++)
    p[j] = q[j] - r[j];
Cout << "Finish\n";
```

Seung-Jai Min

Scott B. Baden /CSE 260/ Winter 2014
Workload decomposition

- Translator automatically generates local loop bounds
- Also inserts any needed barriers
- We use private/shared pragmas to distinguish thread private from global variable
- Decomposition can be static or dynamic
- Dynamic assignment for irregular problems

```cpp
#pragma omp parallel private(i) shared(n)
{
    #pragma omp for
    for(i=0; i < n; i++)
       work(i);
}
```
Parallelizing a nested loop with OpenMP

• We parallelize the outer loop index, indicating shared and private (local) variables

```
#pragma omp parallel private(i) shared(n)
#pragma omp for
   for(i=0; i < n; i++) {
      for(j=0; j < n; j++) {
         u\text{new}[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i, j+1])/4
      }
}
```

• Some implementations can parallelize inner loops

```
mymin = 1 + ($TID * n/nprocs), mymax = mymin + n/nprocs - 1
for(i=mymin; i < mymax; i++)
   for(j=0; j < n; j++)
      u\text{new}[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i, j+1])/4
Barrier();
```
Variable scoping

- Any variables declared outside a parallel region are shared by all threads
- Variables declared inside the region are private
- Used `shared` and `private` declarations to override the defaults

```c
double c = 1 / 6.0, h = 1.0, c2 = h * h;
double ***c = ...;
for (it= 0; it<nIters; it++) {
    #pragma omp parallel shared(U,Un,b,nx,ny,nz,c2,c) private(i,j,k)
    #pragma omp for
        for (int i=1; i<=nx; i++)
            for (int j=1; j<=ny; j++)
                for (int k=1; k<=nz+1; k++)
                    Un[i][j][k]=c* (U[i-1][j][k] + U[i+1][j][k] + U[i][j-1][k] + U[i][j+1][k] +
                                    U[i][j][k-1] + U[i][j][k+1] - c2*b[i-1][j-1][k-1]);
    Swap U ←Un;
}
An application: Matrix Vector Multiplication

\[
\begin{array}{cccc}
  a_{00} & a_{01} & \cdots & a_{0,n-1} \\
  a_{10} & a_{11} & \cdots & a_{1,n-1} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
  \vdots & \vdots & & \vdots \\
  a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\end{array}
\]

\[
\begin{array}{c}
x_0 \\
x_1 \\
\vdots \\
x_{n-1} \\
\end{array}
\]

= \[
\begin{array}{c}
y_0 \\
y_1 \\
\vdots \\
y_{m-1} \\
\end{array}
\]

\[
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}
\]
Initialization

- Allocate and initialize storage outside a parallel region
- In some apps, we initialize in parallel

```c
double **A;
A = new double [N*N];
assert(A);

for ( i=0; i<N; i++ )
  for ( j=0; j<N; j++ )
    A[i*N+j] = 1.0 / (double) (i+j-1);
```
double *A, *x, *y;                              // GLOBAL

// Start timer
    double t0 = -getTime();

#pragma omp parallel shared(A,x,N)
    for (int k = 0; k < reps; k++)
#pragma omp for
        for (i=0; i < N; i++){
            y[i] = 0.0;
            for (j=0; j < N; j++)
                y[i] += A[i*n+j] * x[j];
        }

// Take time
    t0 += getTime();
Reductions in OpenMP

- What if we need to reduce a set of values down to a single value?
- We need to tell OpenMP to ensure atomicity when updating sum, to avoid the race condition

```c
#pragma omp parallel reduction(+:sum)
    for (int i=0; i < N-1; i++)
        gsum += f(x[i]);

i0 = $TID*n/$nthreads, i1 = i0 + n/$nthreads;
for (i=i0, localSum=0; i < i1; i++)
    localSum += f(x[i]);
gsum.atomicAdd(localSum);
```
Race conditions

• The program usually runs correctly
  * Failing to employ synchronization in multithreaded code doesn’t guarantee safety violations, it just allows them!

• But sometimes results are incorrect: non-deterministic

• There was a conflict when updating a shared quantity: a data race
  * Arises when there is at least one writer on shared data
  * The timing of accesses to shared data can affect the outcome

Scott B. Baden /CSE 260/ Winter 2014
Race conditions

- A *Race* condition arises when the timing of accesses to shared memory can affect the outcome.

- Consider this statement, assume $x = 0$
  
  ```
x=x+1;
  ```

- Generated code
  - $r_1 \leftarrow (x)$
  - $r_1 \leftarrow r_1 + \#1$
  - $r_1 \rightarrow (x)$

- Possible interleaving with two threads
  
  **P1**
  
  - $r_1 \leftarrow x$
  - $r_1 \leftarrow r_1 + \#1$
  - $r_1 \rightarrow (x)$
  - $x \leftarrow r_1$

  **P2**
  
  - $r_1 \leftarrow x$
  - $r_1 \leftarrow r_1 + \#1$
  - $r_1 \rightarrow (x)$
  - $x \leftarrow r_1$

  - $r_1(P1)$ gets 0
  - $r_2(P2)$ also gets 0
  - $r_1(P1)$ set to 1
  - $r_1(P1)$ set to 1
  - $r_1(P1)$ set to 1
  - $P1$ writes its $R1$
  - $P2$ writes its $R1$
Avoiding race conditions

- Usually we want to avoid non-determinism
- If we compute with the same inputs we want to obtain the same results
- Not necessarily true for operations that have *side effects* (global variables, I/O and random number generators)
- We need to take steps to avoid race conditions through appropriate program synchronization
  - Atomic variables
  - Critical sections
  - Barriers
- Very difficult to detect atomicity errors
Barriers in OpenMP

• Even if we avoid the race condition through atomicity, we still need the barrier
• Why?

```c
#pragma omp parallel reduction(+:sum)
  for (int i=0; i< N-1; i++)
    gsum += f(x[i]);
Print gsum

i0 = $TID* n/ nthreads, i1 = i0 + n/ nthreads;
for (i=i0, localSum=0; i < i1; i++)
  localSum += f(x[i]);
gsum.atomicAdd(localSum);
Print gsum
```
Dealing with loop carried dependences

• OpenMP will dutifully parallelize a loop when you tell it to, even if doing so “breaks” the correctness of the code

```c
int* fib = new int[N];
    fib[0] = fib[1] = 1;
#pragma omp parallel for num_threads(2)
for (i=2; i<N; i++)
    fib[i] = fib[i-1]+ fib[i-2];
```

• Sometimes we can restructure an algorithm, as we saw in odd/even sorting

• OpenMP may warn you when it is doing something unsafe, but not always
OpenMP is also an API

```c
#ifdef _OPENMP
#include <omp.h>

int nthreads = 1;
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf("Number of openMP threads: %d\n", nthreads);
    }
}
#endif
```
In class exercises – for OpenMP
Questions

1. Iteration to thread mapping
2. Removing data dependencies
3. Dependence analysis
4. Tree Summation
1. Iteration to thread mapping

```c
#pragma omp parallel shared(N, iters) private(i)
#pragma omp for
for (i = 0; i < N; i++)
    iters[i] = omp_get_thread_num();
```

N = 9, # of openMP threads = 3
0 0 0 1 1 1 2 2 2

N = 16, # of openMP threads = 4, schedule(static,2)
0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3

N = 9: 0 0 1 1 2 2 0 0 1

N = 16, # of openMP threads = 4, schedule(dyanmic,2)
3 3 0 0 1 1 2 2 3 3 3 3 3 3 3 3
2 2 3 3 0 0 1 1 2 2 2 2 2 2 2

In $PUB/Examples/OpenMP/Assign$
2. Removing data dependencies

- B initially:  0 1 2 3 4 5 6 7
- B on 1 thread:  7 7 7 7 11 12 13 14
- How can we split into 2 loops so that each loop parallelizes, the result it correct?

```c
#pragma omp for shared (N,B)
for i = 0 to N-1
  B[i] += B[N-1-i];
```
Splitting a loop

• For iterations $i = \frac{N}{2} + 1$ to $N$, $B[N-i]$ reference newly computed data
• All others reference “old” data
• $B$ initially: 0 1 2 3 4 5 6 7
• Correct result: 7 7 7 7 11 12 13 14

for $i = 0$ to $N-1$

$B[i] += B[N-i];$

for $i = \frac{N}{2} + 1$ to $N-1$

$B[i] += B[N-1-i];$

for $i = 0$ to $N/2-1$

$B[i] += B[N-1-i];$

In $\text{PUB/Examples/OpenMP/Assign}$

Compile with $\text{omp}=1$ on “make” line
3. Loop Dependence Analysis

• Which loop(s) can we correctly parallelize with OpenMP?

1. for $i = 1$ to $N-1$

2. for $i = 0$ to $N-2$

3. for $i = 1$ to $N-1$ step 2

4. for $i = 0$ to $N-2$
   $A[i] = B[i]$;
   $E[i] = C[i+1]$;
   }

Scott B. Baden /CSE 260/ Winter 2014 33
Fin