Lecture 4
CSE 260 – Parallel Computation
(Fall 2015)
Scott B. Baden

Multithreading
OpenMP
Using Bang

• Do not use Bang’s *front end* for heavy computation
• Use batch, or interactive nodes, via *qlogin*
• Use the front end for editing & compiling only
Today’s lecture

• Autotuning
• Multithreading
• Programming with OpenMP
2x2 Matmul with SSE instrinsics

#include <emmintrin.h>

void square_dgemma (int N, double* A, double* B, double* C){
  __m128d c1 = _mm_loadu_pd( C+0*N);  //load unaligned block in C
  __m128d c2 = _mm_loadu_pd( C+1*N);
  for( int i = 0; i < 2; i++ ){
    __m128d a1 = _mm_load1_pd( A+i+0*N);  // load i-th column of A (A0i,A0i)
    __m128d a2 = _mm_load1_pd( A+i+1*N);  // [i=0/1] (A1i,A1i)
    __m128d  b = _mm_load_pd( B+i*N);    // load aligned i-th row of B
    c1 = _mm_add_pd( c1, _mm_mul_pd( a1, b ) );  // rank-1 update
    c2 = _mm_add_pd( c2, _mm_mul_pd( a2, b ) );
  }
  _mm_storeu_pd( C+0*N, c1 );  //store unaligned block in C
  _mm_storeu_pd( C+1*N, c2 );
}

\[ C_{00} C_{01} \mathbf{+=} \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix} \]

\[
\begin{align*}
C_{00} \mathbf{+} & = A_{00}B_{00} + A_{01}B_{10} \\
C_{10} \mathbf{+} & = A_{10}B_{00} + A_{11}B_{10} \\
C_{01} \mathbf{+} & = A_{00}B_{01} + A_{01}B_{11} \\
C_{11} \mathbf{+} & = A_{10}B_{01} + A_{11}B_{11}
\end{align*}
\]
Autotuning

• Performance tuning is complicated and involves many code variants
• Performance models are not accurate, for example, in choosing blocking factors
  ♦ Conflict misses
  ♦ Too hard for a human being to sort through all the options

• Autotuning
  ♦ Let the computer do the heavy lifting: generate and measure program variants and the parameter search space
  ♦ ATLAS [used in Matlab]
A search space

A 2-D slice of a 3-D register-tile search space. The dark blue region was pruned. (Platform: Sun Ultra-Ii, 333 MHz, 667 Mflop/s peak, Sun cc v5.0 compiler)

Jim Demmel

Scott B. Baden / CSE 260, UCSD / Fall '15
Today’s lecture

- Autotuning
- Multithreading
- Shared memory hierarchy
  - Cache coherence
  - False sharing
Multithreading

- On shared memory, a natural model for programming is called *multithreading*

- Programs execute as a set of *threads*
  - Threads are usually assigned to different physical cores
  - Each thread runs the same code as an independent instruction stream

- Threads communicate implicitly through shared memory (e.g. the heap), but have their own private stacks

- They coordinate (synchronize) via shared variables
What is a thread?

• A thread is similar to a procedure call with notable differences
• The control flow is different
  ♦ A procedure call is “synchronous;” return indicates completion
  ♦ A spawned thread executes asynchronously until it completes, and hence a return doesn’t indicate completion
• A new storage class: shared data
  ♦ Synchronization may be needed when updating shared state (thread safety)

![Diagram showing shared memory and private memory]

\[ y = \text{..} s \text{..} \]
Which of these storage classes can never be shared among threads?

A. Globals declared outside any function
B. Local automatic storage
C. Heap storage
D. Class members (variables)
E. B & C
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
Software Execution model

• We will assume the **Same Program Multiple Data** programming model = “SPMD”
  ✷ Other names: “loosely synchronous” or “bulk synchronous”
  ✷ All threads run the same program
  ✷ The number of threads is specified (and fixed) at the time we run the program
• It is possible to have dynamic thread creation, but we will not use this parallel control flow
• Each thread has an associated *index*, a unique integer
Parallel control flow

- Parallel program
  - 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 threads
• POSIX Threads “standard” (pthreads):
  IEEE POSIX 1003.1c-1995
    ♦ Low level interface
    ♦ Beware of non-standard features
• OpenMP – program annotations
• Java threads – not used in high performance computation
• Parallel programming languages
  ♦ Co-array FORTRAN
  ♦ UPC
OpenMP

• A higher level interface for threads programming
• Parallelization handled via source code annotations
• An evolving standard: http://www.openmp.org
• Higher level than explicit threads, though with less flexibility
• 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);
}
```

```c
i0 = $TID*n/$nthreads;
i1 = i0 + n/$nthreads;
for (i=i0; i < i1; 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
  • All threads wait until all have arrived. Then they may leave the barrier
  • After leaving the barrier the team is disbanded. Only the initial thread continues, other enter a wait state
• Thread teams can be created and disbanded many times during, but this can be costly
• A clever compiler can avoid so many thread creations and joins
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";
Workload decomposition

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

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

```c
i0 = $TID\times n/nthreads;
i1 = i0 + n/nthreads;
for (i=i0; i< i1; i++)
    work(i);
```
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
```
Loop Dependence Analysis

• MPI cannot parallelize a loop unless it is free to reorder the iterations arbitrarily.

• If the result of one iteration depends on another, then we have a *loop carried dependence*; OpenMP cannot parallelize such loops.

• Which loop(s) cannot be parallelized?

1. for $i = 1$ to $N-1$
   \[ A[i] = A[i] + B[i-1]; \]

2. for $i = 0$ to $N-2$
   \[
   \begin{align*}
   A[i] &= B[i]; \\
   C[i] &= A[i] + B[i]; \\
   E[i] &= C[i+1]; \\
   \end{align*}
   \]

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

4. for $i = 0$ to $N-2$
   \[ A[i+1] = A[i] + 1; \]
Other kinds of data dependencies

• Consider the following loops

```c
#pragma omp parallel
{
#pragma omp for
for (int i=0; i< N-1; i++)
    a[i] = (b[i+1] – b[i-1])/2h
#pragma omp for
for (int i=N-1; i>=0; i--)
    b[i] = (a[i+1] – a[i-1])/2h
}
```

• Why are results incorrect?
Other kinds of data dependencies

• Consider the following loops

```c
#pragma omp parallel
{#pragma omp for
  for (int i=1; i< N-1; i++)
    a[i] = (b[i+1] - b[i-1])/2h

#pragma omp for
  for (int N-2; i>0; i--)
    b[i] = (a[i+1] - a[i-1])/2h
}
```

• Why are results incorrect?

Results will be incorrect because the array `a[ ]`, in loop #2, depends on the outcome of loop #1 (a true dependence) and we don’t know when the threads finish

• What else can cause a problem?
**Barrier Synchronization**

- To deal with true- and anti-dependences, OpenMP inserts a barrier between loops:

```
for (int i=0; i< N-1; i++)
    a[i] = (b[i+1] - b[i-1])/2h

BARRIER
```

```
for (int i=N-1; i>=0; i--)
    b[i] = (a[i+1] -a[i-1])/2h
```

- No thread may pass the barrier until all have arrived
- Do we need the barrier in this case? **Yes**

```
for (int i=0; i< N-1; i++)
    a[i] = (b[i+1] - b[i-1])/2h

BARRIER?
```

```
for (int i=N-1; i>=0; i--)
    c[i] = a[i]/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 $i=N/2+1$ 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 $i = N/2+1$ to $N-1$
\[ B[i] += B[N-1-i]; \]

In $PUB/Examples/OpenMP/Assign$
Compile with omp=1 on “make” line
Parallelizing a loop nest with OpenMP

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

```c
#pragma omp parallel private(i) shared(n)
#pragma omp for
  for(i=0; i < n; i++)
    for(j=0; j < n; j++) {
      u_{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/nproc), mymax = mymin + n/nproc - 1
for(i=mymin; i < mymax; i++)
  for(j=0; j < n; j++)
    u_{new}[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1])/4
Barrier();
```

• Generated code
  * An implicit barrier after the loop
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

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

- The matrix is 2D array, but is memory is 1 dimensional
- We will assume *row major* order
  (Fortran uses column major)

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

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

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

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

• Allocate and initialize storage outside a parallel region
• In some applications, we initialize in parallel in order to optimize for NUMA

```cpp
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);
```
Computation

double *A, *x, *y;

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

#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 += get_time();
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

• Sometimes a program may exhibit non-deterministic behavior: **results are not correct**

• This can happen when there is 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
**Under the hood of a data race**

- Consider this statement, assume \( x == 0 \)
  \[ x = x + 1; \]

- Generated code
  - \( r1 \leftarrow (x) \)
  - \( r1 \leftarrow r1 + \#1 \)
  - \( r1 \rightarrow (x) \)

- Possible interleaving with two threads

  \[
  \begin{align*}
  P1 & \quad P2 \\
  r1 & \leftarrow x & r1 & \leftarrow x & r1(P1) \text{ gets 0} & r2(P2) \text{ also gets 0} \\
  r1 & \leftarrow r1 + \#1 & r1 & \leftarrow r1 + \#1 & r1(P1) \text{ set to 1} & r1(P1) \text{ set to 1} \\
  x & \leftarrow r1 & x & \leftarrow r1 & P1 \text{ writes its R1} & P2 \text{ writes its R1}
  \end{align*}
  \]
Avoiding data races

• 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
• *Failing to employ synchronization in multithreaded code doesn’t guarantee safety violations, it just allows them!*
Barriers in OpenMP

- Even if we avoid the race condition through an atomic update, we still need the barrier
- Why?

```cpp
#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
```
Incorrect behavior

• 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];
```

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