Lecture 7

OpenMP:
Reduction, Synchronization,
Scheduling & Applications

Announcements

- Section and Lecture will be switched on Thursday and Friday
- Thursday: section and Q2
- Friday: Lecture

© 2010 Scott B. Baden / CSE 160 / Winter 2011

Today's lecture

- Reduction, Synchronization, Scheduling & Applications
- Introduction to OpenMP

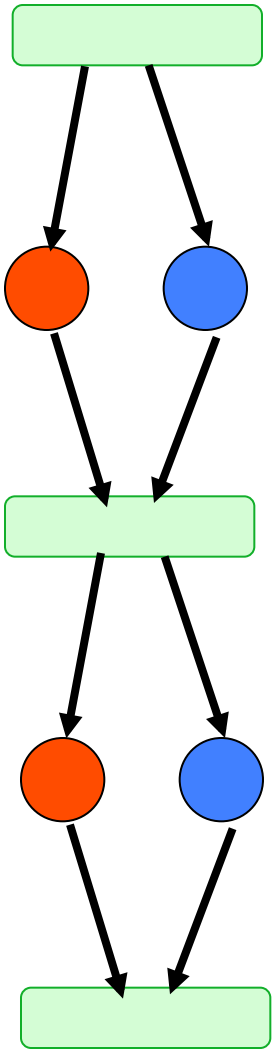
© 2010 Scott B. Baden / CSE 160 / Winter 2011

OpenMP under the hood

- A program begins life as a single “boss” thread
- When the boss encounters a parallel construct, it creates a team of worker 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 and are dissolved; they enter a wait state
  - Only the master thread continues,
- Thread teams can be created and dissolved many times during program execution

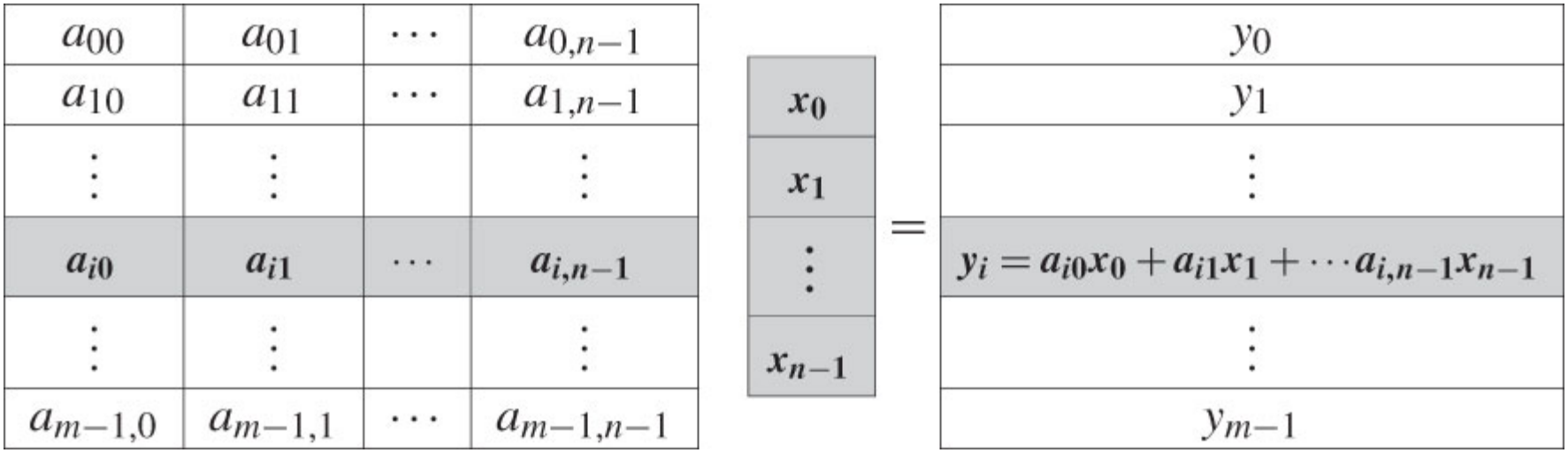
www.ncsa.uiuc.edu/UserInfo/Resources/Software/Intel/Compilers/10.0/main_cls/mergedProjects/optaps_cls/whskin_homepage.htm

- A clever compiler can avoid so many thread creations and joins



© 2010 Scott B. Baden / CSE 160 / Winter 2011

Matrix Vector Multiplication



© 2010 Scott B. Baden / CSE 160 / Winter 2011

Matrix Vector Multiplication in OpenMP

- We can query the number of threads in OpenMP, within a parallel region

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

int NT =-99999997;

#pragma omp parallel
{
#pragma omp single
    // Only one thread needs to query the # threads
      NT = omp_get_num_threads();
}
```

© 2010 Scott B. Baden / CSE 160 / Winter 2011

Initialization

- We allocate and initialize storage outside a parallel region

```
double **A;
A = (double**) \
   malloc(sizeof(double*)*N + sizeof(double)*N*N);
assert(A);

for(j=0;j<N;j++) A[j] = (double*)(A+N) + j*N;

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

© 2010 Scott B. Baden / CSE 160 / Winter 2011

Computation

```
 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][j] * x[j];
      }

   // Take time
   t0 += getTime();
```

© 2010 Scott B. Baden / CSE 160 / Winter 2011

Compare with pthreads coding

- "Outline" the computation into a thread procedure
- Spawn and join threads
- Partition the rows of the matrix over processors

© 2010 Scott B. Baden / CSE 160 / Winter 2011

Thread management

- The boss thread also takes on a worker's task

```
for(int64_t t=1;t<NT;t++)
    assert(!pthread_create(&thrds[t], NULL, matvec_thr,  …));
    int zero = 0;

    matvec_thr(reinterpret_cast<void *>(zero));

    // Join the threads
    for(int64_t t=1;t<NT;t++){
        assert(!pthread_join(thrds[t],  NULL));
    }
```

© 2010 Scott B. Baden / CSE 160 / Winter 2011

Data partitioning

- Partitioning is the process of splitting up the computation: how we map iterations to threads
- By default, OpenMP partitions the outermost loop only, into intervals
- Each processor gets N/NT elements
- What if N % NT ~= 0?
  - Let q = N%NT
  - First q processors get ⌈N/NT⌉ elements, others get ⌊N/NT⌋ elements

P0 P1 P2 P3

© 2010 Scott B. Baden / CSE 160 / Winter 2011

The thread function

```
    int chunk= N/N, q = N%NT;
    int low = (TID < q ) ? TID * (chunk+1) \
                         : rem*(chunk+1)+(TID-rem)
*chunk;
    int    hi = (TID < q ) ? low + chunk+1
                         : low+chunk;
// Repeat several times to get reasonable timings
    for (int k = 0; k<reps; k++)
        for (int i=low; i<hi; i++){
            y[i] = 0.0;
            for (int jj=0; j<N; j++)
                y[i] += A[i][j] * x[j];
```

© 2010 Scott B. Baden / CSE 160 / Winter 2011

Sections – Task Parallelism

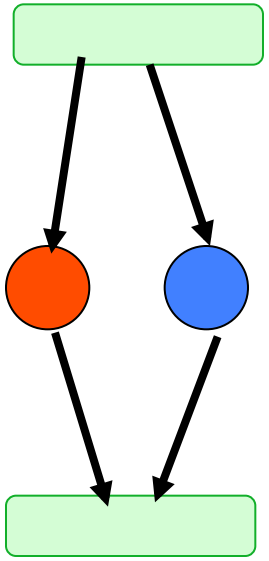
```
#pragma omp parallel // Begin a parallel construct
{ // form a team
  // Each team member executes the same code
  #pragma omp sections    // Begin work sharing
  {
    #pragma omp section    // A unit of work
    {functionA(..);}

    #pragma omp section    // Another unit
    {functionB(..);}

  } // Wait until both units complete

} // End of Parallel Construct; disband team

// continue serial execution
```

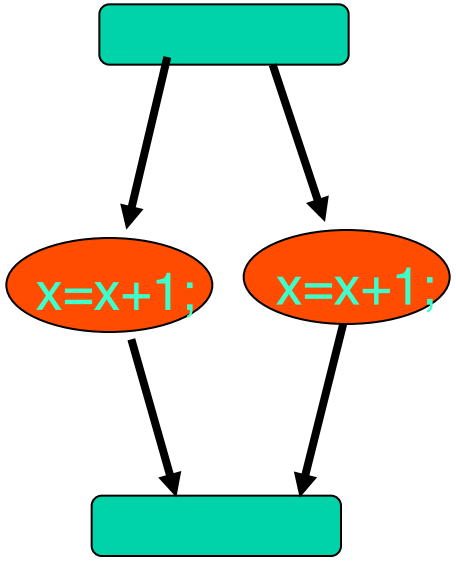


© 2010 Scott B. Baden / CSE 160 / Winter 2011

Critical Sections

- Only one thread at a time may run the code in a critical section
- Uses mutual exclusion to implement critical sections

```
#pragma omp parallel // Begin a parallel construct
{
  #pragma omp sections    // Begin worksharing
  { //
    #pragma omp critical    // Critical section
    {x = x + 1}
    #pragma omp critical    // Another critical section
    {x = x + 1}
  ... // More Replicated Code
   #pragma omp barrier // Wait for all members to arrive
     } // Wait until both units of work complete
}
```



© 2010 Scott B. Baden / CSE 160 / Winter 2011

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;
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]);
  U ↔Un;
}
```

Reductions in OpenMP

- OpenMP uses a local accumulator, which it then accumulates globally after the loop finishes

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

i0 = $TID*n/NT , i1 = i0 + n / NT;
for (i=i0, localSum=0; i < i1; i++)
   localSum += x[i];
```

All threads accumulate localSum into Global Sum

© 2010 Scott B. Baden / CSE 160 / Winter 2011

Reduction

```
double c = 1 / 6.0, err=0;
#pragma omp parallel shared(U,B,c)
#pragma omp for reduction(+:err)
for (int i=1; i<=nx; i++)
  for (int j=1; j<=ny; j++)
    for (int k=1; k<=nz; k++){
      double du = 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] - 6.0*B[i-1][j-1][k-1]);
      double = B[i-1][j-1][k-1] - du;
      err = err +  r*r;
   }
double resid = sqrt(err)/(double)((nx+2)*(ny+2)*(nz+2));
```

© 2010 Scott B. Baden / CSE 160 / Winter 2011

Dealing with Data Dependences

Loop carried dependences

- OpenMP will dutifully paralllize a loop when you tell it to, even if doing so "breaks" the correctness of the code

```
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 in odd/even sorting

© 2010 Scott B. Baden / CSE 160 / Winter 2011

Parallel Sorting

- Sorting is fundamental algorithm in data processing
  - Given an unordered set of keys $x_0, x_1, \ldots, x_{N-1}$
  - Return the keys in sorted order
- The keys may be character strings, floating point numbers, integers, or any object for which the relations >, <, and == hold
- We'll assume integers here
- Will talk about other algorithms later on

© 2010 Scott B. Baden / CSE 160 / Winter 2011

Compare and exchange sorts

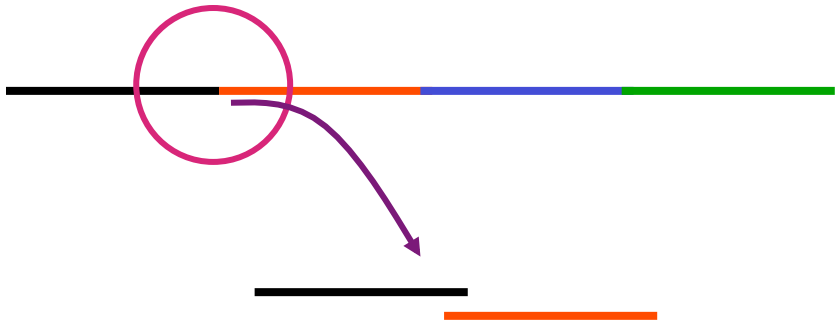
- Simplest sort, based on the bubble sort algorithm
- The fundamental operation is compare-exchange
- Compare-exchange(a[j] , a[j+1])
  - swaps its arguments if they are in decreasing order
  - satisfies the post-condition that $a[j] \leq a[j+1]$
  - Returns FALSE if a swap was made

```
for i = N-1 to 1 by -1 do
    done = TRUE;
    for j = 0 to i-1 do // Compare-exchange(a[j] , a[j+1])
        if (a[i] < a[j])  { a[i] ⇔ a[j];
                            done=FALSE; }
    end do
    if (done) break;
 end do
```

Loop carried dependencies

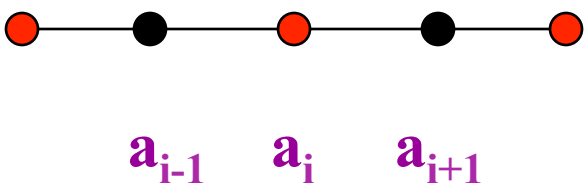
- We cannot parallelize bubble sort owing to the *loop carried dependence* in the inner loop
- The value of a[j] computed in iteration j depends on the a[i] computed in iterations 0, 1, …, j–1

```
for i = N-1 to 1 by -1 do
    done = TRUE;
    for j = 0 to i-1 do
        done = Compare-exchange(a[j] , a[j+1])
    end do
    if (done) break;
end do
```



Odd/Even sort

- If we re-order the comparisons we can parallelize the algorithm
  - number the points as even and odd
  - alternate between sorting the odd and even points
- This algorithm parallelizes since there are no loop carried dependences
- All the odd (even) points are decoupled



© 2010 Scott B. Baden / CSE 160 / Winter 2011

The algorithm

```
done = false;

for i = 0 to n-1 do

   for j = 0 to n-1 by 2 do   // Even
       done &= Compare-exchange(a[j] , a[j+1]);
   end do

   for j = 1 to n-1 by 2 do // Odd
       done &= Compare-exchange(a[j] , a[j+1]);
   end do
   if (done) break;

end do
```

© 2010 Scott B. Baden / CSE 160 / Winter 2011

Odd/Even Sort Code

```
int OE = lo % 2;      Global int OE;
for (s = 0; s < MaxIter; s++) {
    bsync();
     int done = Sweep(Keys, OE, lo, hi, NT, TID);    /* Odd phase */

     bsync();
     done &= Sweep(Keys, 1-OE, lo, hi, NT, TID);  /* Even phase */

     int allDone = AllDone(done,TID);

     if (allDone){
          s++;
          break;
     }

} /* End For */
```

```
bsync();
if (!TID)
   allDone = 1;
bsync();
pthread_mutex_lock(&mutex);
    allDone &= (done);
pthread_mutex_unlock(&mutex);
bsync();
```

© 2010 Scott B. Baden / CSE 160 / Winter 2011

Inside Sweep

```
int     Sweep(int *Keys, int OE, int lo, int hi, int NT, int TID){
    int Hi = hi;
    if (TID == NT-1)
        Hi--;
    int done = 1;
    for (int i = OE+lo; i <= Hi; i+=2) {
      if (Keys[i] > Keys[i+1]){
          Keys[i] ↔ Keys[i+1];
          done = 0;
      }
    }
   return done;
}
```

© 2010 Scott B. Baden / CSE 160 / Winter 2011

In class problem solving

Time constrained scaling

- Sum N numbers on P processors
- Let N >> P
- Determine the largest problem that can be solved in time $T=10^4$ time units on 512 processors
- Let time to perform one addition = 1 time unit
- Let β = time to add a value inside a critical section

Performance model

- Local additions: N/P - 1
- Reduction: β (lg P - 1)
- Since N >> P

  $T(N,P) \sim (N/P) + \beta (\lg P - 1)$
- Determine the largest problem that can be solved in time T= 10^4 time units on P=512 processors, β = 1000 time units
- Constraint: $T(512,N) \leq 10^4$

  $\Rightarrow (N/512) + 1000 (\lg 512 - 1) = (N/512) + 1000*(8) \leq 10^4$

  $\Rightarrow \quad N \leq 1 \times 10^6$ (approximately)

© 2010 Scott B. Baden / CSE 160 / Winter 2011