

CSE 160

Lecture 9

Load balancing
Parallel Sorting

Announcements

- Quiz on Friday
- Signup for Friday labs sessions in APM

Today's lecture

- Load balancing the N-body method with dynamic scheduling
- Parallel Sorting (I)
- In class exercises
 - Some today, the rest on Wednesday
 - At end of today's slides

Particle simulation

- Two routines contain the work to be parallelized according to **openmp #pragmas**
- What's not inside **OMP parallel for** should run serially
- Synchronization?

```
void SimulateParticles(){
    for( int step = 0; step < nsteps; step++ ) {
        apply_forces(particles,n);

        if (imbal)
            imbal_particles(particles,n);

        move_particles(particles,n);

        VelNorms(particles,n,uMax,vMax,uL2,vL2);
    }
}
```

Code for computing the force

- OpenMP parallelization of the outermost loop

```
void apply_forces( particle_t* particles, int n){  
    #pragma omp parallel for shared(particles,n)  
    for( int i = 0; i < n; i++ ) {  
        particles[i].ax = particles[i].ay = 0;  
        if ((particles[i].vx  $\neq$  0) || (particles[i].vy  $\neq$  0)) // for imbalanced  
            for (int j = 0; j < n; j++ ) { // case  
                if (i==j)  
                    continue;  
                particles[i].{ax,ay} += coef * {dx,dy};  
            }  
        }  
    }  
}
```

Load Balancing

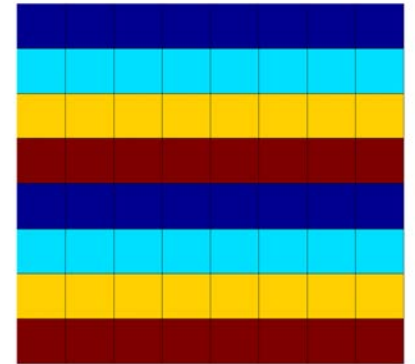
- The starter code has the `-b` option to shut off the force computation for the first $n/4$ particles
- A uniformly blocked workload assignment will result in a load imbalance
 - Some processors finish later than other
 - **Load balancing efficiency** $\eta = T_1 / (P T_p)$
 - $\eta < 1.0$
- 2 Alternatives
 - Cyclic mapping (static used in A1)
 - Dynamic scheduling

First approach: BLOCK CYCLIC

- Divide into pieces of size CHUNK
- Processor k gets chunks 0, $NT*CHUNK$, $2*NT*CHUNK$, ...
- Also called *round robin* or *block cyclic*

```
#pragma omp parallel for shared(particles,n) \  
    schedule(static, CHUNK)
```

```
for( int i = 0; i < n; i++ ) {  
    particles[i].ax = particles[i].ay = 0;  
    if ((particles[i].vx ≠ 0) || (particles[i].vy ≠ 0))  
        for (int j = 0; j < n; j++ ) {  
            if (i==j)  
                continue;  
            particles[i].{ax,ay} += coef * {dx,dy};  
        }  
    }  
}
```

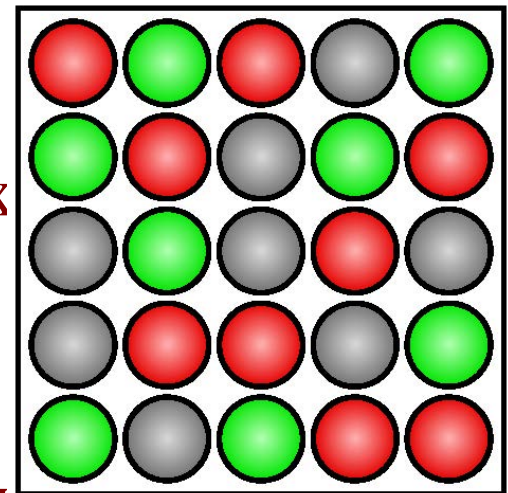


Second approach: dynamic scheduling

- Unlike CYCLIC decomposition, workload assignments are made on demand
- We subdivide workload into pieces of size **CHUNK**
- Also called *processor self scheduling*

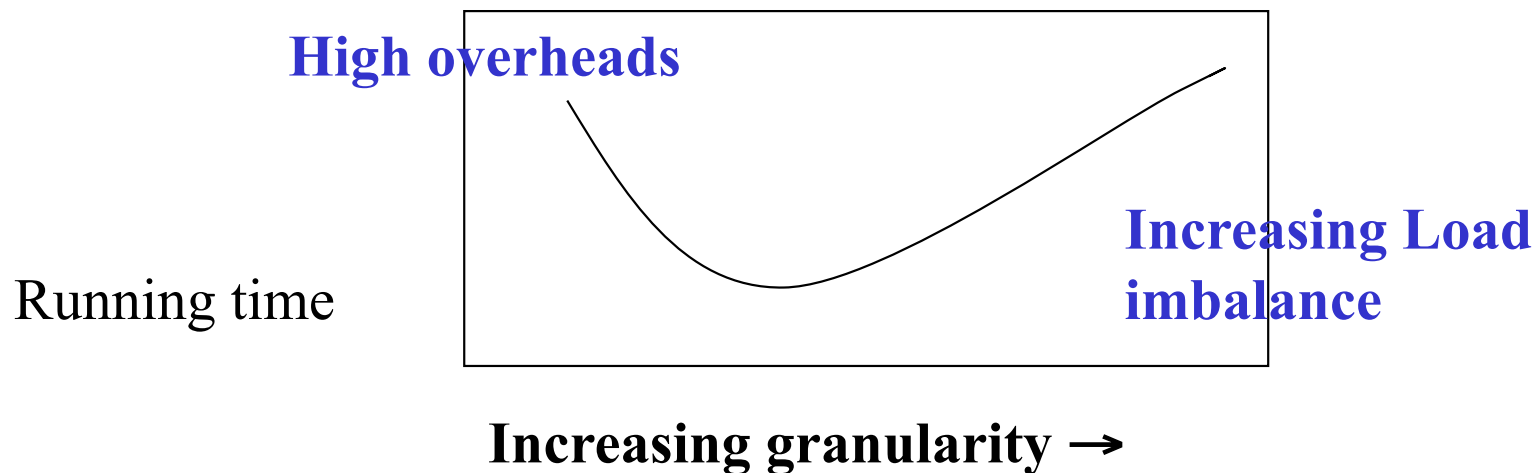
```
#pragma omp parallel for shared(particles,n) \  
    schedule(dynamic, CHUNK)
```

```
for( int i = 0; i < n; i++ ) {  
    particles[i].ax = particles[i].ay = 0;  
    if ((particles[i].vx != 0) || (particles[i].vy != 0))  
        for (int j = 0; j < n; j++ ) {  
            if (i==j)  
                continue;  
            particles[i].{ax,ay} += coef * {dx,dy},  
        }  
    }  
}
```



Tradeoffs in dynamic scheduling

- Dynamic workload assignments: processors schedule themselves
- Each thread samples a unique (and disjoint) set of indices, changes from run to run
- A shared counter or work queue represents the work
- User tunes work granularity (chunk size) trading off the overhead of workload assignment against increased load imbalance
 - Finest granularity: each point is a separate task
 - Coarsest granularity: one block per processor



How does self scheduling work?

```
SelfScheduler S(n, P, chunk);  
  
while ( S.getChunk(mymin, mymax))  
    for i = mymin to mymax  
        work(i, chunk);  
    end for  
end while
```

Implementation

- Critical section can be costly, but OMP atomic is restricted to simple update statements, e.g `++` `+=`
- C++ `atomic<T>` not guaranteed to be lock free, but probably more efficient
- Useful member functions in `atomic::`

```
boolean getChunk(int& mymin, int& mymax ){ // Inefficient
#pragma omp critical // uses locks
{
    k = __counter;
    __counter += __chunk;
}
if (__counter > (__n - __chunk)
    return false;

mymin = k, mymax = k + __chunk;
return true;
}
```

OpenMP is also an API

```
#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
```

Today's lecture

- Load balancing the N-body method with dynamic scheduling
- **Parallel Sorting (I)**

Parallel Sorting

- Sorting is fundamental algorithm in data processing
 - Given an unordered set of keys $x_0, x_1, \dots, x_{N-1}$
 - 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

Compare and exchange sorts

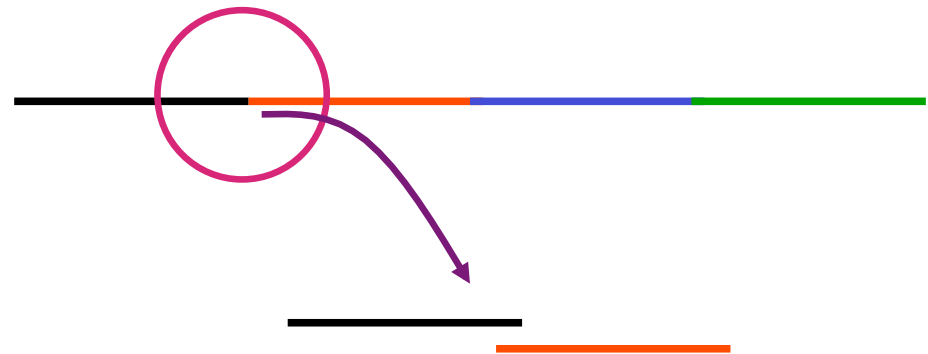
- Simplest sort, based on the bubble sort algorithm
- The fundamental operation is compare-exchange
- **Compare-exchange($a[j]$, $a[j+1]$)**
 - swaps its arguments if they are in decreasing order
 - satisfies the post-condition that $a[j] \leq a[j+1]$
 - returns FALSE if a swap was made

```
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 ( $a[i] < a[j]$ ) {  $a[i] \leftrightarrow a[j]$ ;
                        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 $0, 1, \dots, j-1$

```
#pragma omp parallel for shared(N, a)
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
```



Dealing with loop carried dependences

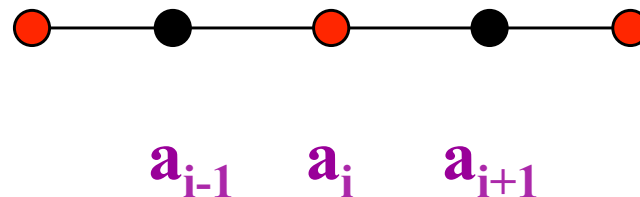
- OpenMP will dutifully parallelize a loop when you tell it to, even if doing so “breaks” the code (incorrect results)

```
int* fib = new int[N];  
    fib[0] = fib[1] = 1;  
#pragma omp parallel for shared(N, fib)  
    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*

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



The algorithm

```
done = false;
```

```
for i = 0 to n-1 do
```

```
    #pragma omp parallel for shared(n, a)
```

```
    for j = 0 to n-1 by 2 do // Even
```

```
        done &= Compare-exchange(a[j] , a[j+1]);
```

```
    end do
```

```
    #pragma omp parallel for shared(n, a)
```

```
    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
```

Odd/Even Sort Code

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

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

    if (done){
        s++;
        break;
    }
} /* End For */
```

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];
```

- Equivalent threads code

```
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

Using the reduction clause

- More efficient than using a critical section
- In C++, we use <atomic>

```
int Sweep(int *Keys, int OE, int lo, int hi){
int done = 1;
#pragma omp parallel for shared(done,Keys) private(l,lo,hi) \
                        reduction(*:done)
    for (int i = OE+lo; i <= Hi; i+=2) {
        if (Keys[i] > Keys[i+1]){
            Keys[i] ↔ Keys[i+1];
            done *= 0;
        }
    }
return done;
}
```

In class exercises

Questions

1. Iteration to thread mapping
2. Removing data dependencies
3. Dependence analysis
4. Time constrained scaling
5. Tree Summation
6. Performance

1. Iteration to thread mapping

```
#pragma omp parallel shared(N, iters) private(i)
```

```
#pragma omp for
```

```
for (i = 0; i < N; i++)
```

```
    iters[i] = omp_get_thread_num();
```

In \$PUB/Examples/OpenMP/Assign
Compile with omp=1 on “make” line

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(dynamic,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 2

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 is correct?

#pragma omp for shared (N,B)

for i = 0 to N-1

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

B[0] += B[7], B[1] += B[6], B[2] += B[5]

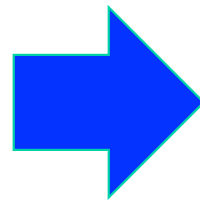
B[3] += B[4], B[4] += B[2], B[5] += B[2]

B[7] += B[0]

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 $i = 0$ to $N/2-1$
 $B[i] += B[N-1-i];$
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

3. Loop Dependence Analysis

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

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

2. for $i = 0$ to $N-2$
 $A[i+1] = A[i] + 1;$

3. for $i = 0$ to $N-1$ step 2
 $A[i] = A[i-1] + A[i];$

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

4. Time constrained scaling

- Sum N numbers on P processors
- Let $N \gg P$
- 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: $\beta (\lg P)$
- Since $N \gg P$
$$T(N,P) \sim (N/P) + \beta (\lg P)$$
- Determine the largest problem that can be solved in time $T = 10^4$ time units on $P = 512$ processors, $\beta = 1000$ time units
- Constraint: $T(512,N) \leq 10^4$
$$\Rightarrow (N/512) + 1000 (\lg 512)$$
$$= (N/512) + 1000 \cdot (9) \leq 10^4$$
$$\Rightarrow N \leq 5 \times 10^5 \text{ (approximately)}$$

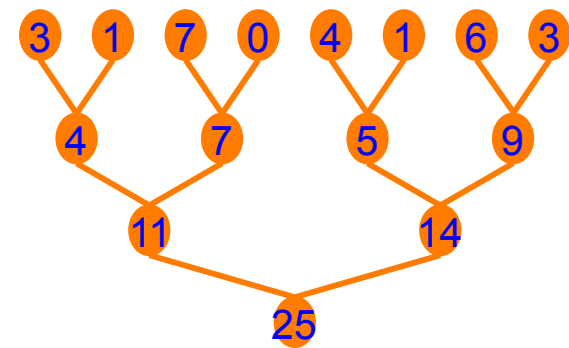
5. Tree Summation

- Input: an array $x[]$, length $N \gg P$
- Output: Sum of the elements of $x[]$
- Goal: Compute the sum in $\lg P$ time

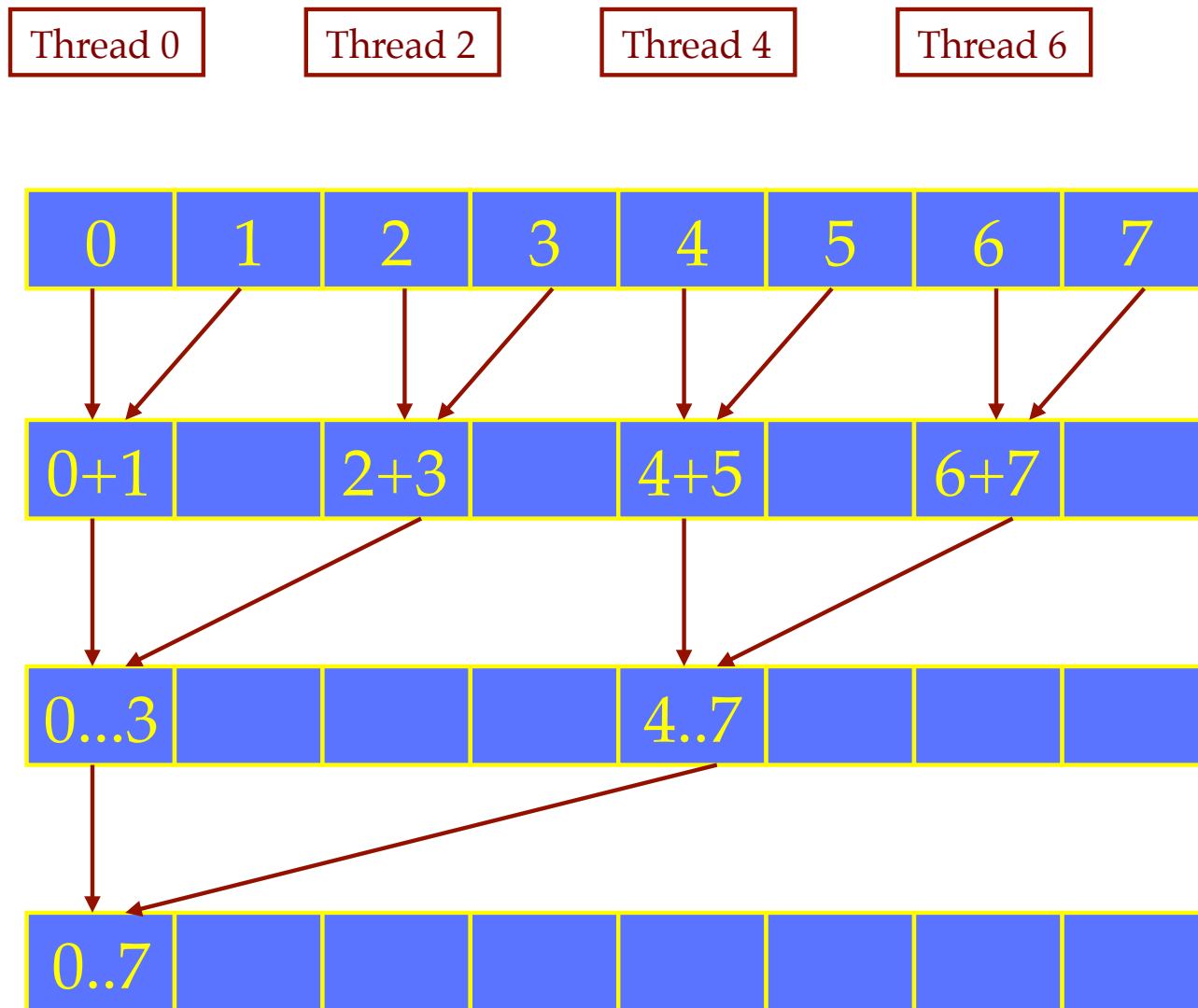
```
sum = 0;  
for i=0 to N-1  
    sum += x[i]
```

- Assume P is a power of 2, $K = \lg P$
- Starter code
for $m = 0$ to $K-1$ {

}
}



Visualizing the Summation



6. Performance

- You observe the following running times for a parallel program running a fixed workload N
- Assume that the only losses are due to serial sections
- What is the speedup and efficiency on 8 processors?
- What will the running time be on 4 processors?
- What is the maximum possible speedup on an infinite number of processors?
- What fraction of the total running time on 1 processor corresponds to the serial section?
- What fraction of the total running time on 2 processors corresponds to the serial section?

NT	Time
1	10000
2	6000
8	3000