Problem

- Have a program
- Want it to run faster

Solution

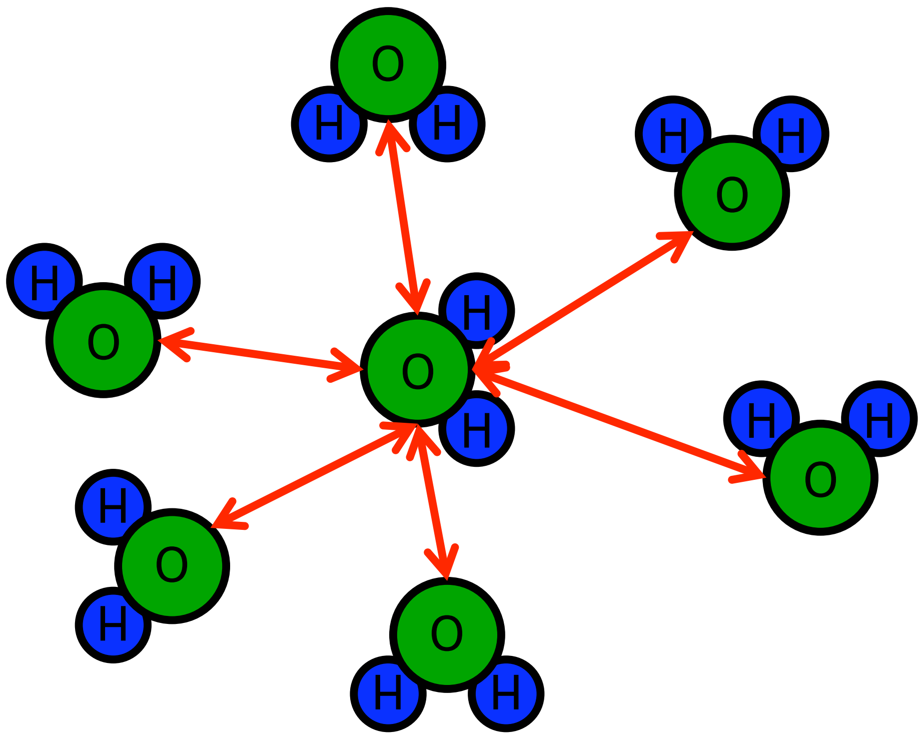
- Profile program
- Find loops that take most time
- Parallelize each loop, run parallel program
  - No synchronization between parallel iterations
  - Measure quality of service, performance
  - Fix any quality of service, performance problems
    - Introduce synchronization
    - Privatize data
    - Replicate data
  - Use memory profiling, goal-directed approach

Quality of Service

- Number that measures relative difference between output from serial and parallel programs
- Smaller is better; zero indicates serial and parallel programs produce same output
- Quality of service of 0.10 indicates that parallel program produces output that differs from serial output by 10%
- We can run this basic approach without the quality of service concept – it will work even if we require the parallel program to produce the identical output as the serial program

Example

- Water computation (Jade benchmark suite)
- Simulates liquid water molecules (N^2 algorithm)



Execution Time Profiling

- Most of time spent in two loops: interf (63%), poteng (36%)
- Parallelize interf loop, run program (8 threads):

Speedup	Quality of Service
1.501	0.385
1.515	0.383
1.522	0.382
1.518	0.387
1.515	0.399
1.505	0.393
1.514	0.391
1.509	0.379

What Is Going Wrong?

- Potential quality of service problems (that we can do something about)
  - Atomicity violations
  - Interference on data that should be thread local
  - Data dependence violations
- Potential performance problems (that we can do something about)
  - True sharing between threads
  - False sharing between threads
  - Locality problems

How To Fix Problems?

- Get memory profiling information
  - Run program, generate trace containing
  - Loop iteration entry and exit events
  - Memory access events
    - Accessed address, type of access (read, write)
    - Instruction performing the access
- Consider each problem in turn
  - Come up with a potential fix for that problem (guided by memory profiling information)
  - Try the fix and see if it works!

Sample Trace

iter 0	iter 1	iter 2	iter 3
…	…	…	…
k : rd(a_1)	k : rd(a_3)	k : rd(a_1)	k : rd(a_2)
l : wr(a_1)	l : wr(a_3)	l : wr(a_1)	l : wr(a_2)
…	…	…	…
k : rd(a_2)	k : rd(a_4)	k : rd(a_3)	k : rd(a_4)
l : wr(a_2)	l : wr(a_4)	l : wr(a_3)	l : wr(a_4)
…	…	…	…
i : rd(b)	i : rd(b)	i : rd(b)	i : rd(b)
j : wr(b)	j : wr(b)	j : wr(b)	j : wr(b)

Sample Trace Structure

iter 0	iter 1	iter 3	iter 4
…	…	…	…
$k : rd(a_1)$	$k : rd(a_3)$	$k : rd(a_1)$	$k : rd(a_2)$
$l : wr(a_1)$	$l : wr(a_3)$	$l : wr(a_1)$	$l : wr(a_2)$
…	…	…	…
$k : rd(a_2)$	$k : rd(a_4)$	$k : rd(a_3)$	$k : rd(a_4)$
$l : wr(a_2)$	$l : wr(a_4)$	$l : wr(a_3)$	$l : wr(a_4)$
…	…	…	…
$i : rd(b)$	$i : rd(b)$	$i : rd(b)$	$i : rd(b)$
$j : wr(b)$	$j : wr(b)$	$j : wr(b)$	$j : wr(b)$

2 Patterns

$k{:}rd(\alpha), l{:}wr(\alpha)$
pattern
(8 instances)

$i{:}rd(\beta), j{:}wr(\beta)$
pattern
(4 instances)

Synchronization Insertion for Atomicity

- Look at memory trace for interf loop
- See lots of patterns like **κ : rd(α); λ : wr(α);**
- Hypothesis is that lack of atomicity is problem
- Fix is to insert lock operations
  - **κ : rd(α); λ : wr(α);** becomes
  - **acquire(α); κ : rd(α); λ : wr(α); release(α);**

Prioritizing Synchronization Insertion

- Multiple patterns in trace
- Synchronize patterns one at a time (until result is acceptable)
- Prioritize according to interference density

$$\text{Priority} = \frac{\displaystyle\sum_{\text{All Instances of Pattern}} \begin{array}{c}\text{Number of Instructions that}\\ \text{Interfere with that Instance}\end{array}}{\text{Number of Instances of Pattern}}$$

Sample Trace Structure

iter 0	iter 1	iter 3	iter 4
...	...	...	...
k : rd(a_1)	k : rd(a_3)	k : rd(a_1)	k : rd(a_2)
l : wr(a_1)	l : wr(a_3)	l : wr(a_1)	l : wr(a_2)
...	...	...	...
k : rd(a_2)	k : rd(a_4)	k : rd(a_3)	k : rd(a_4)
l : wr(a_2)	l : wr(a_4)	l : wr(a_3)	l : wr(a_4)
...	...	...	...
i : rd(b)	i : rd(b)	i : rd(b)	i : rd(b)
j : wr(b)	j : wr(b)	j : wr(b)	j : wr(b)

2 Patterns

k:rd(α), l:wr(α)
pattern
(8 instances)
Priority = 1

i:rd(β), j:wr(β)
pattern
(4 instances)
Priority = 3

Synchronization Results

- Synchronize **i:rd(β), j:wr(β)** pattern first

Speedup	Quality of Service
0.491	0.00065
0.457	0.00048
0.455	0.00080
0.493	0.00038
0.446	0.00085
0.492	0.00044
0.494	0.00063
0.491	0.00030

- Quality of service good, performance bad

Next Hypothesis

- Performance is bad because of true sharing
- Replicate shared data
- Transform
  - **acquire(α); κ : rd(α); λ : wr(α); release(α);**
  - **t = lookup(α); κ : rd(t); λ : wr(t);**
- **lookup** accesses local replica of address **α**
  - Hash table maps global addresses to local replicas
  - Initialize each replica to zero when initially accessed
  - Add replicas together at end of loop
  - Store sum in global address

Results After Replication

Speedup	Quality of Service
1.78	0.0044
1.77	0.0041
1.78	0.0034
1.80	0.0045
1.79	0.0040
1.81	0.0043
1.80	0.0043
1.77	0.0047

- Good performance, good quality of service
- So keep these transformations
- Move on to next loop (poteng)

Parallelization of poteng Loop

Parallelize poteng loop

Speedup	Quality of Service
2.35	0.645
2.29	0.641
2.38	0.649
2.35	0.645
2.43	0.655
2.38	0.649
2.32	0.647
2.36	0.649

Synchronize three $\kappa : rd(\alpha); \lambda : wr(\alpha);$ patterns

Speedup	Quality of Service
0.698	0.0036
0.703	0.0038
0.695	0.0036
0.691	0.0046
0.708	0.0042
0.690	0.0039
0.691	0.0043
0.688	0.0048

Replicate three $\kappa : rd(\alpha); \lambda : wr(\alpha);$ patterns

Speedup	Quality of Service
5.29	0.0044
5.48	0.0042
4.82	0.0037
5.26	0.0052
5.02	0.0048
4.32	0.0041
5.36	0.0047
5.37	0.0046