Arrays, queues and linked lists

”Problems that are relatively easy to parallelize”

Trivially Parallelizable Problems

Characteristics:

- Independent computations
- Each computation of (approximately) the same size
- Easy to partition the computations

What to keep in mind:

- Partition computation on the highest possible level
- Is it worth it? Overhead vs. speedup
- <u>Shared memory:</u> Careful with false sharing (cache effects)
- <u>Distributed memory:</u> Need enough computation to offset cost of communication

Matrix Operations

C = A + B where A, B, and C are *n* x *n* matrices

```
for i = 1,n
  for j = 1,n
    c(i,j) = a(i,j) + b(i,j)
```

$t_s = n^2,\ t_p = n^2/p + t_{comm}$

OpenMP:

- Parallelize outermost loop
- Little risk of false sharing
- Different access pattern might change how matrices are stored

Message passing on distributed memory computer

Speedup depends on where A and B are stored initially and

where C should be stored.

A, B, and C on process 0:

- t_p will most likely be higher than t_s

A, B, and C distributed:

- $t_{comm} = 0$

Dynamic Scheduling

Setting

- Unknown job sizes
- Different processor speeds
- Unknown number of jobs

Shared memory

- Use guided or dynamic load balancing
- Can also use `task` directive

Distributed memory

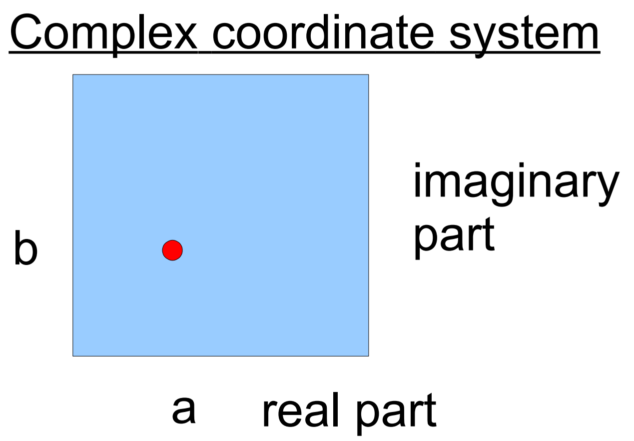
- Master thread sends out tasks upon request
- Each worker thread receives tasks, computes, and then sends back answer.
- The job size must be determined empirically

Example: Recursive Functions

Recursive functions $s(n) = f(s(n-1), s(n-2) \ldots s(0))$

Fibonacci sequence: $s(n) = s(n-1) + s(n-2)$, where $s(0) = 0$, $s(1) = 1$

2-dimensional functions:
Takes a two-dimensional point as input (a,b)



Complex numbers:
$c = a + bi$ where $i = \sqrt{-1}$
a is the *real* part, b is the *imaginary* part

$s(n) = (s(n-1))^2 + c$ where $s(0) = 0$

Question: For which values of c will s(n) diverge and for which will the value stay bounded?

The Mandelbrot Set

$s(n) = (s(n-1))^2 + c$ where $c = a + bi$ and $s(0) = 0$

If s(n) does not diverge then c is in the Mandelbrot set.

Fact: If $S(n) = x + yi$ then $s(n)$ will diverge iff $|s(n)| = \sqrt{(x^2 + y^2)} \geq 2$.

$$\begin{aligned}(s(n))^2 &= (x+yi)^2 = x^2 + 2xyi + y^2i^2 \\ &= x^2 - y^2 + 2xyi\end{aligned}$$

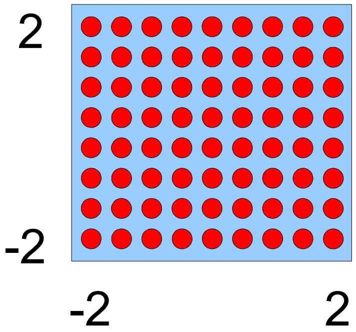
$$\begin{aligned}s(n+1) &= (s(n))^2 + a + bi \\ &= \underbrace{(x^2 - y^2 + a)}_{\textit{Real part}} + \underbrace{(2xy + b)i}_{\textit{Imaginary part}}\end{aligned}$$

```
while  (i < limit) and (x² + y²) < 4
   {t = (x² - y² + a);  y = (2xy + b); x = t; i++;}

if i == limit
   then (a,b) is in the M.S.
```

Plotting the Mandelbrot Set

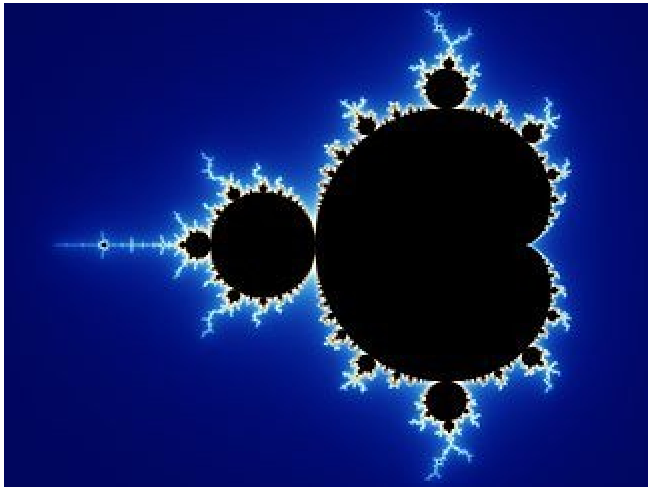
Area of computation is continuous
→ Need to discretize.



Speed of convergence varies
→ Number of computations depends on coordinates

Use colors to indicate speed of divergence

<u>Going parallel</u>

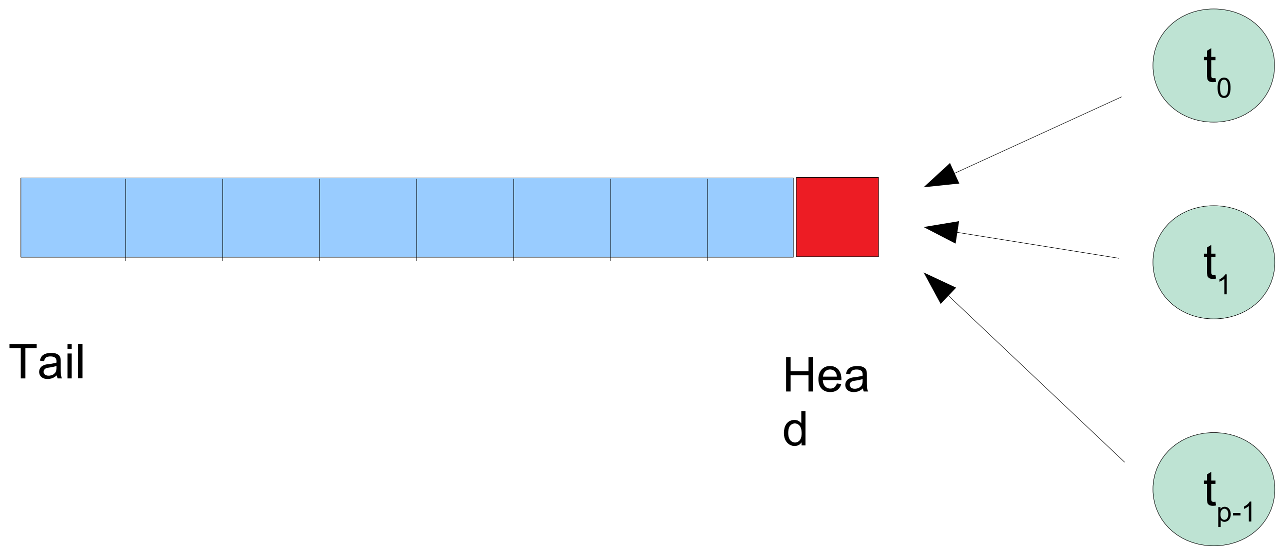


Computation in each point is independent
→ Easy to parallelize with shared memory

Distributed memory
→ Partitioning of the work only requires
coordinate information
→ O(1) data out.
O((n^2 * max it.)/p) computations
O(n^2/p) data in.

Queues

Parallel access to a common queue can be difficult



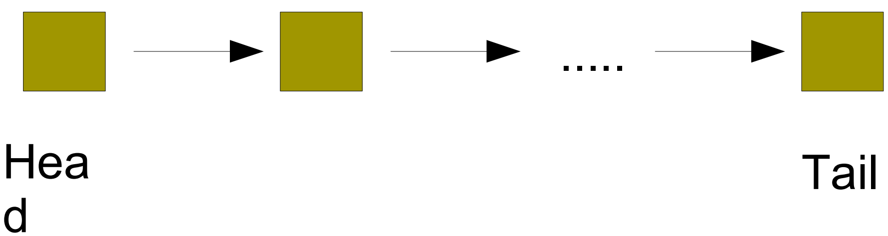
Must serialize access to Head (or Tail)
A thread needs to both read and write to Head pointer

Solutions:
- Use a software lock (omp_lock)
- Use hardware dependent methods (Compare-and-swap)
- Could try with one queue for each thread?

Still, introduces sequential part...

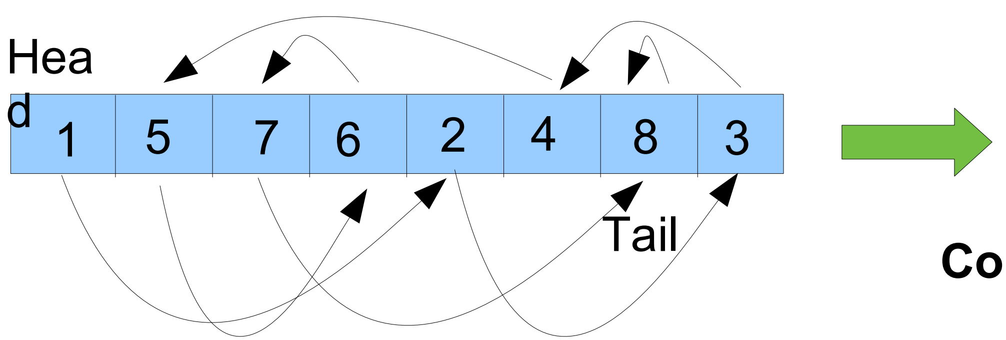
Lists

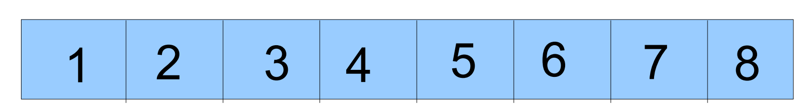
General case: Can only access Head (and maybe Tail)



Can only be accessed sequentialy :-(

But what if the list is stored in an array?



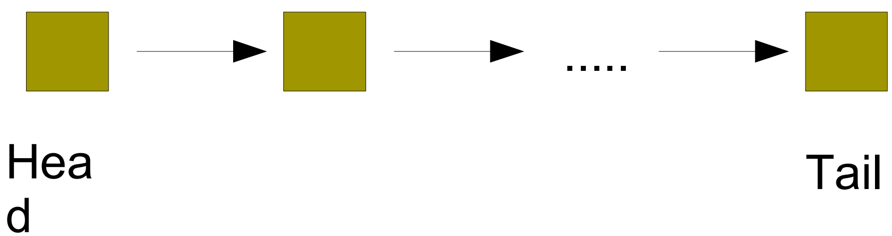


Could we sort the elements in parallel?

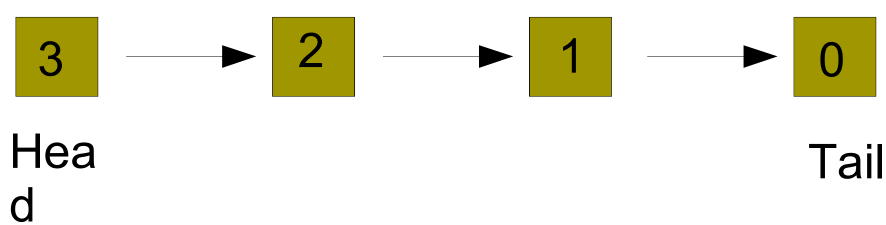
Might be more suitable for parallel processing.

List Ranking Problem

Need to compute where to put each element in the final array



For each element count the number of ensuing elements.
Can easily compute final position and move elements in parallel.



(See book or later lecture for actual algorithm)

Monte Carlo Simulations

Monte Carlo methods tend to follow a particular pattern:

- Define a domain of possible inputs.
- Generate inputs randomly from a probability distribution over the domain.
- Perform a deterministic computation on the inputs.
- Aggregate the results.

Usefull for search problems where finding an exact value is too costly

A Simple Example

Given n numbers a_1, a_2, ... a_n.

Problem: Find a number larger than the median.

Traditional algorithm: Must look at at least n/2 + 1 values.

Thus takes time O(n)

Monte Carlo Method:

Choose k values at random

Pick max value out of the k and return.   Time O(k)

How good is it?

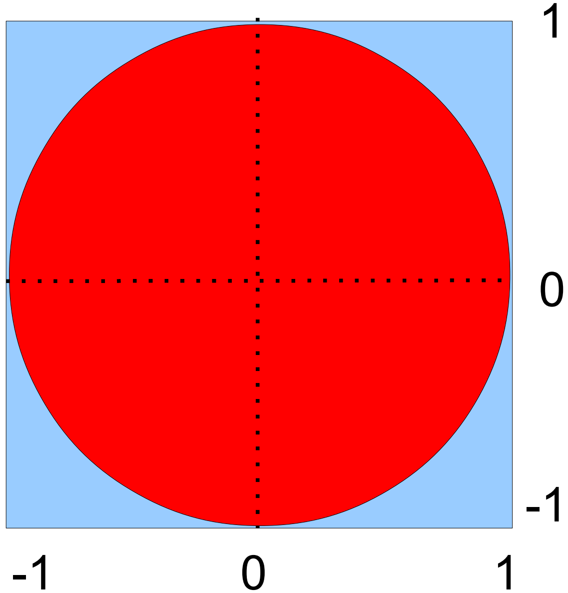
P(max value in upper half) = 1 – P(max value in lower half)

= 1 – P(all k values in lower half)

= $1 - (0.5)^k$

k = 10 gives P(success) > 0.999, k = 20 gives P(success) > 0.999999

Calculating π



Area of circle = $\pi \times r^2 = \pi$
Area of square = 4

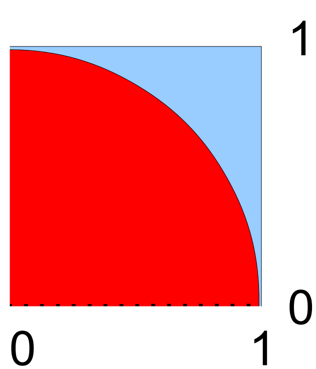
Area of circle / Area of square = $\pi / 4$

Choose random point in the square
Register number of points inside of the circle.

(x,y): $x^2 + y^2 > 1 \rightarrow$ Outside of circle
$x^2 + y^2 \leq 1 \rightarrow$ Inside of circle

Number inside / Total number $\rightarrow \pi/4$

Easy to parallelize!!



Could also use only one quadrant
Area of circle = $\pi/4$
Area of square = 1

Generating random numbers

Pseudorandom numbers:

- Recursion: $x_{i+1} = f(x_i)$, $x_0 = seed$
  - f is chosen such that "$x_1, x_2, ..., x_n$ appear to be random"
  - Guessing x_{n+1} should be "difficult" when x_0 is unknown
- Reproducible sequence:
  - Use a deteministic seed
- Unpredictable sequence:
  - Use a hash function value and current time as seed
- `int random(void); // Returns the next pseudorandom number`
- `void srandom(unsigned int seed); // Sets the seed`

Linear congruential generators (Knuth 1981):

- Recursion: $x_{i+1} = (ax_i+c)\%m$
  - Parameters **a**, **c** and **m** should be chosen to give the sequence a long period
  - After one period, the sequence will repeat itself.

Generating random numbers

Pseudorandom numbers in parallel programs:

- Centralized: A dedicated thread generates all pseudorandom numbers
  - May involve many requests to the generating thread
  - Not very suitable for Monte Carlo simulation
- Distributed: Each thread generates the numbers as they are needed
  - Sequences must be distinct
  - Seeds must be distinct
  - Can use a hash function of (time and) process rank

srandom(),random(): Sequential methods for seeding and generating random numbers, all numbers are generated from the same sequence.

srandom_r(), random_r(): Thread safe, generates different sequences on different threads. Slightly more complicated to use.