Parallel Computing I

SMP: Load Balancing and Reduction

Looking Back, Looking Forward

Last two weeks:

- Why parallel computing?
- Parallel program designs
- Massively parallel problems
- SMP parallel programs with Parallel Java
    - parallel teams
    - parallel for loops
- Performance metrics

This week:

- load balancing
- reduction

Load balancing and Reduction

Even when the opportunities for parallelism are obvious,
some parallel programs require careful design to acheive good speedups.

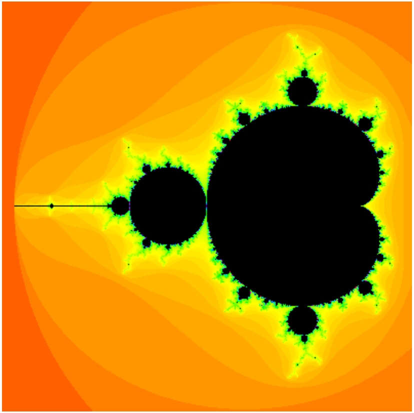
Issues and trade-offs relating to *load balancing*:

- study a program that generates images of the Mandelbrot set

Issues and trade-offs relating to *reduction*:

- study a program that calculates π using a Monte Carlo algorithm
- study a program that generates histograms of the Mandelbrot set

Mandelbrot Set



Invented/Discovered by Benoit Mandelbrot (1924 – 2010) at IBM in 1977; now perhaps the most famous fractal object.

Mandelbrot Set

The Mandelbrot set is a set of points in the complex plane, defined as follows:

- For each point in the plane $c = x + yi$, compute a sequence of points $z_n = a_n + b_n i$:
  - $z_0 = 0 + 0i$
    - $a_0 = 0$ and $b_0 = 0$
  - $z_{n+1} = z_n^2 + c$
    - $a_{n+1} = a_n^2 - b_n^2 + x$ and $b_{n+1} = 2 * a_n * b_n + y$

Mandelbrot Set

The Mandelbrot set is a set of points in the complex plane, defined as follows:

- For each point in the plane $c = x + yi$, compute a sequence of points $z_n = a_n + b_n i$:
    - $z_0 = 0 + 0i$
        - $a_0 = 0$ and $b_0 = 0$
    - $z_{n+1} = z_n^2 + c$
        - $a_{n+1} = a_n^2 - b_n^2 + x$ and $b_{n+1} = 2 * a_n * b_n + y$
- If there exists a k such that $|z_n| < k$ for all n (that is, if the sequence remains bounded), then c is a member of the Mandelbrot set.
    - black pixels in previous image
- If $|z_n| \to \infty$ as $n \to \infty$ (that is, if the sequence shoots off to infinity), then c is not a member of the Mandelbrot set.
    - colored pixels in previous image

Computing the Mandelbrot Set

How do we compute whether or not a point is in the Mandelbrot set?

Examining an infinite sequence requires infinite time,
but we want a program that runs in finite time.

Insight: If $|z_n| > 2$ for some n, then the rest of the sequence must shoot off to infinity (and the c that led to this z_n is not a member of the Mandelbrot set).

Computing the Mandelbrot Set

How do we compute whether or not a point is in the Mandelbrot set?

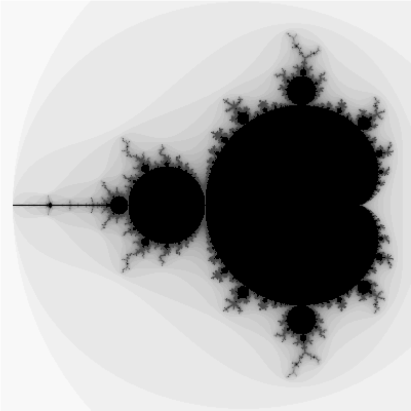
Examining an infinite sequence requires infinite time,
but we want a program that runs in finite time.

Insight: If $|z_n| > 2$ for some n, then the rest of the sequence must shoot off to infinity (and the c that led to this z_n is not a member of the Mandelbrot set).

Compromise:

- Set a limit L on the length of the sequence to examine.
- If $|z_n| > 2$ for some $n \leq L$, then c is definitely not a member.
  - Color c according to smallest n for which $|z_n| > 2$.
- If $|z_n| < 2$ for all $n \leq L$, then c might be a member.
  - Color c black.

Mandelbrot Set



- Points in the set are black.
- Points not in the set are gray, with hue $(\boldsymbol{n_e}/\boldsymbol{L})^{\boldsymbol{r}}$
  - $\boldsymbol{n_e}$ is smallest $\boldsymbol{n} \leq \boldsymbol{L}$ such that $|\boldsymbol{z_n}| > \mathbf{2}$
  - $\boldsymbol{L}$ is the iteration limite
  - $\boldsymbol{r}$ is a scaling parameter

Color Images

To produce an image of the Mandelbrot set,
we need to generate and save an image file.

Lots of frameworks:

- `java.awt.image` package
- `java.awt.Image` class
- `java.awt.BufferedImage` class

Lots of image formats:

- JPEG
- PNG

Pros/Cons for using in a parallel program?

Color Images

Parallel Java provides a (simple) model for images.

Parallel Java Graphics (PJG)

```
// Construct an instance of class edu.rit.image.PJGColorImage
int[][] pixeldata = new int[ROWS][COLS];
PJGColorImage image = new PJGColorImage(ROWS,COLS,pixeldata);

// Specify the color of pixels.
pixeldata[i][j] = IntRGB.pack(redI, greenI, blueI);
pixeldata[i][j] = RGB.pack(redF, greenF, blueF);
pixeldata[i][j] = HSB.pack(hueF, satF, briF);

// Specify the output stream on which to write the PJG file
PJGImage.Writer writer = image.prepareToWrite(outputstream);
writer.write();
writer.close();
```

Pros/Cons for using in a parallel program?

MandelbrotSetSeq.java

code/MandelbrotSetSeq.java

MandelbrotSetSmp.java

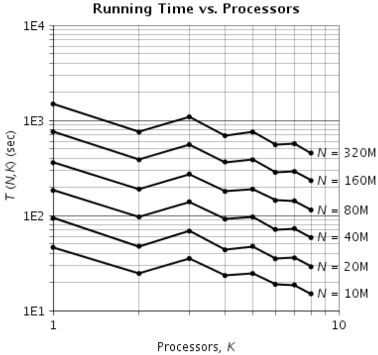
How do we convert the sequential Mandelbrot set program to a parallel Mandelbrot set program?

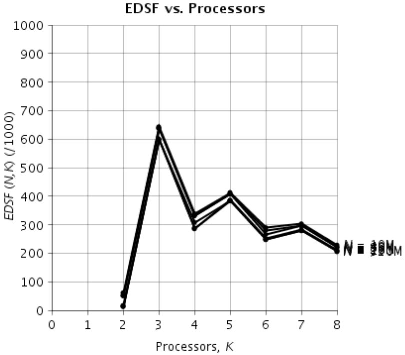
- What portions of the Mandelbrot set program are parallelizable?
- What portions of the Mandelbrot set program are not parallelizable?
- What pattern does the Mandelbrot set use?
- Are there any sequential dependencies?
- Is there any cache interference?

MandelbrotSetSmp.java

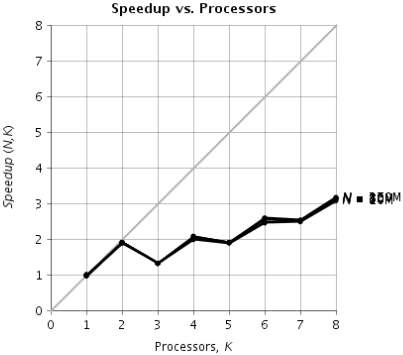
code/MandelbrotSetSmp.java

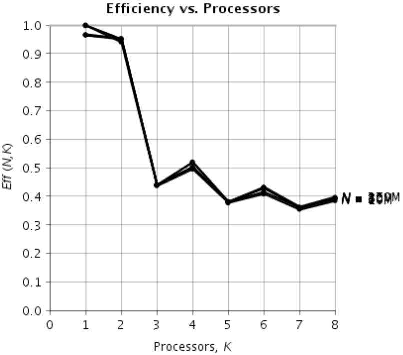
MandelbrotSetSmp Running Time and EDSF



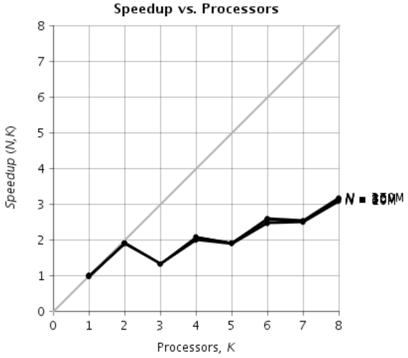


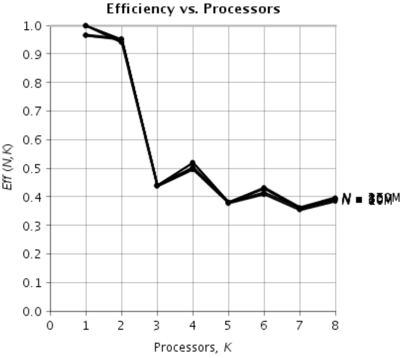
MandelbrotSetSmp Speedup and Efficiency





MandelbrotSetSmp Speedup and Efficiency





What has gone (horribly, horribly) wrong?

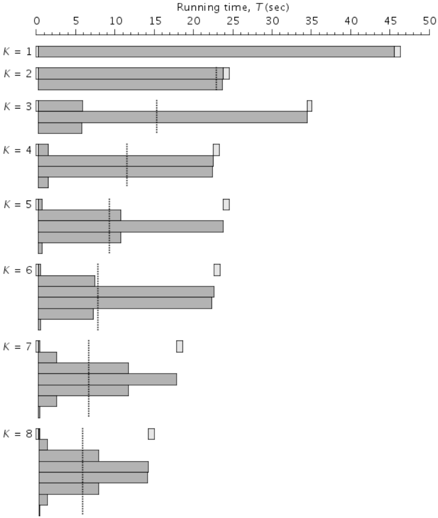
MandelbrotSetSmp2.java

`code/MandelbrotSetSmp2.java`

- records start/finish times of each `ParallelRegion.run()` method
- records start/finish times of each `IntegerForLoop.run()` method

Examine running times with **1**, **2**, **3**, and **4** threads.

Load Balance



Load Balance

- Load Balance (B): the extent to which each thread in a parallel program does the same amount of work.

  $$B = \frac{T_p(K)}{(T_p(1)/K)} = \frac{K \cdot T_p(K)}{T_p(1)}$$

  where $T_p(K)$ is the running time of the program's parallel portion (not the whole program) on K processors.

  - $T_p(K)$ will be the running time of the longest running thread.

Load Balance

- Load Balance (B): the extent to which each thread in a parallel program does the same amount of work.

$$B = \frac{T_p(K)}{(T_p(1)/K)} = \frac{K \cdot T_p(K)}{T_p(1)}$$

where $T_p(K)$ is the running time of the program's parallel portion (not the whole program) on K processors.

  - $T_p(K)$ will be the running time of the longest running thread.

Some questions:

- What is the "ideal" B?
- What is a "good" B?
- What is a "bad" B?
- What is the "worst" B?

Load Balance

K	$T_p(K)$	$T_p(1)$	B
2	23511	45293	1.04
3	34195	45293	2.26
4	22298	45293	1.97
5	23484	45293	2.59
6	22400	45293	2.97
7	17585	45293	2.72
8	13953	45293	2.42

When the load is not balanced, the program takes more time reducing the speedup and efficiency.

Achieving a Balanced Load

AES Key Search vs. Mandelbrot Set

- AES Key Search divided the iterations (key space) equally among threads
- Mandelbrot Set also divided the iterations (rows) equally among threads

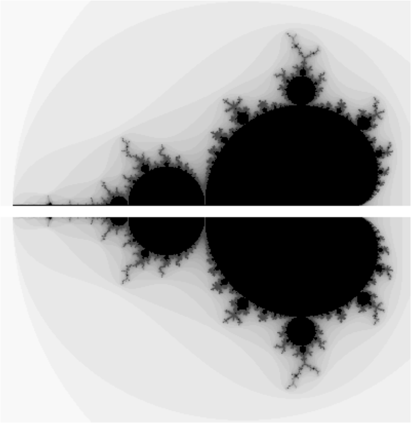
Achieving a Balanced Load

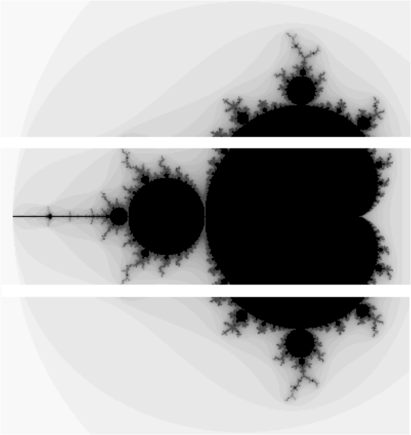
AES Key Search vs. Mandelbrot Set

- AES Key Search divided the iterations (key space) equally among threads
- Mandelbrot Set also divided the iterations (rows) equally among threads

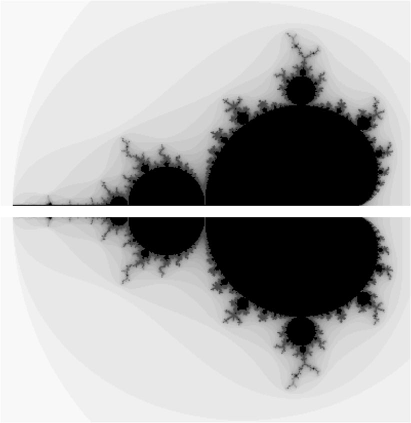
- AES Key Search does (almost exactly) same computation each iteration
- Mandelbrot Set does (very) different calculations each iteration
  - For some points not in the set, iteration terminates quickly
  - For points (maybe) in the set, iteration requires $\mathbf{1000}$ loops

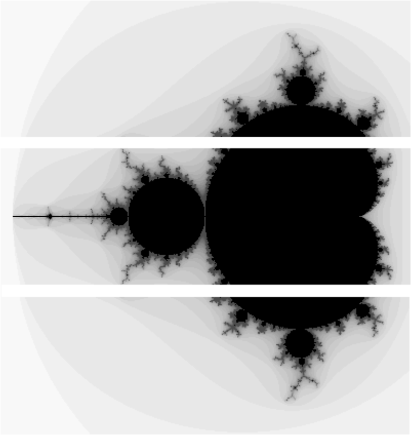
Achieving a Balanced Load





Achieving a Balanced Load





Aside: Number of pixels is not really a good measure of the “size” of the problem, although it is easy to measure.

Achieving a Balanced Load

In order to balance the load of the Mandelbrot Set program, the iterations (rows) should *not* be divided equally among threads.

- Some threads must calculate more rows, the rows with mostly gray pixels
- Some threads must calculate fewer rows, the rows with mostly black pixels

Problem: We know which rows are cheap and which are expensive *after* completing the computation, but we need to unevenly divide the rows *before* starting the computation.

Parallel For Loop Schedules

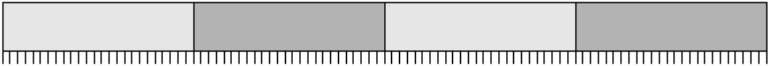
Parallel Java uses/provides *schedules*
to divide the iterations of a parallel for loop unevenly among threads.

Three kinds of schedules:

- Fixed
- Dynamic
- Guided

Fixed Schedule

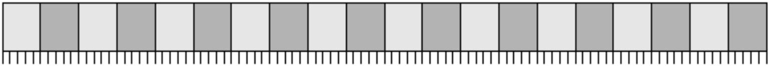
Divide the set of N loop iterations into K chunks (one per thread).



100 iterations, 4 threads

Dynamic Schedule

Divide the set of N loop interations into N/S chunks, each of size S. (Default chunk size is $\mathbf{1}$.)

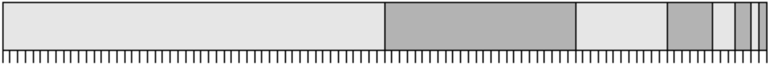


100 iterations, chunk size 5

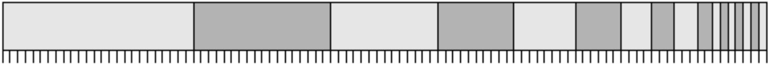
- When a thread's `run` method returns, the thread receives the next available chunk.

Guided Schedule

Divide the set of N loop interations into chunks that get smaller.



100 iterations, 1 thread (chunk sizes: 50, 25, 12, 6, 3, 2, 1, 1)



100 iterations, 2 threads (chunk sizes: 25, 18, 14, 10, 8, 6, 4, 3, 3, 2, 1 × 7)

- The size of each chunk is half the remaining number of iterations divided by the number of threads (except each chunk is no smaller than a specified minimum chunk size (default minimum chunk size is **1**)).

Parallel For Loop Schedules

Which parallel loop schedule should we use?

Want to balance load, while minimizing loop overhead.

- Where/What is the loop overhead?

Parallel For Loop Schedules

Which parallel loop schedule should we use?

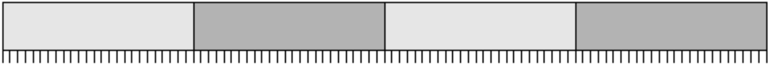
Want to balance load, while minimizing loop overhead.

- Where/What is the loop overhead?
- The more chunks into which the loop iterations are partitioned, the larger the loop overhead

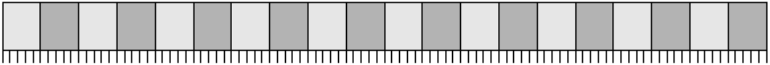
One rule:

- If every loop iteration's computation takes the same amount of time, then use a fixed schedule.
- If some loop iterations' computations take different amounts of time, then use a guided or dynamic schedule.

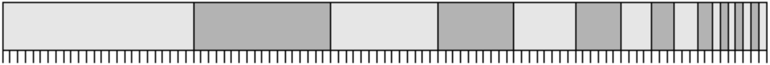
Parallel For Loop Schedules



Fixed schedule: 100 iterations, 4 threads



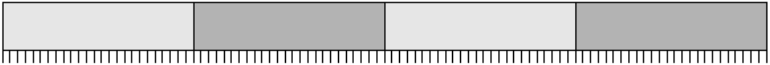
Dynamic schedule: 100 iterations, chunk size 5



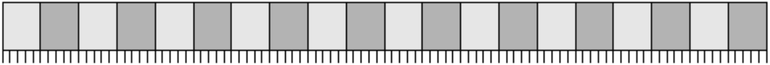
Guided schedule: 100 iterations, 2 threads

What schedule is *guaranteed* to acheive “good” balance,
no matter how long individual loop iteration’s computations take?

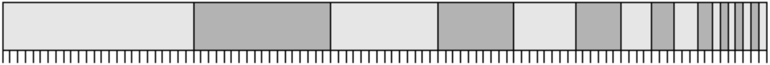
Parallel For Loop Schedules



Fixed schedule: 100 iterations, 4 threads



Dynamic schedule: 100 iterations, chunk size 5



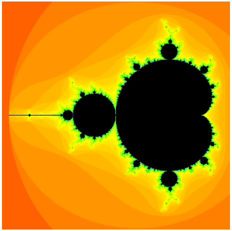
Guided schedule: 100 iterations, 2 threads

What schedule is *guaranteed* to acheive "good" balance,
no matter how long individual loop iteration's computations take?

Why don't we always use this schedule?

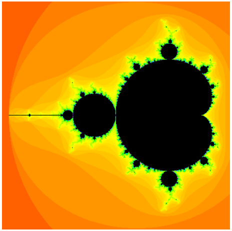
Parallel For Loop Schedules

Experimental data from textbook uses guided schedule for Mandelbrot Set.
Is this a reasonable choice?

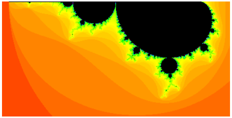


Parallel For Loop Schedules

Experimental data from textbook uses guided schedule for Mandelbrot Set. Is this a reasonable choice?



How would we expect this schedule to perform when generating this image?



Parallel For Loop Schedules

Specifying a parallel for loop's schedule:

- Implement the `IntegerForLoop.schedule()` method:

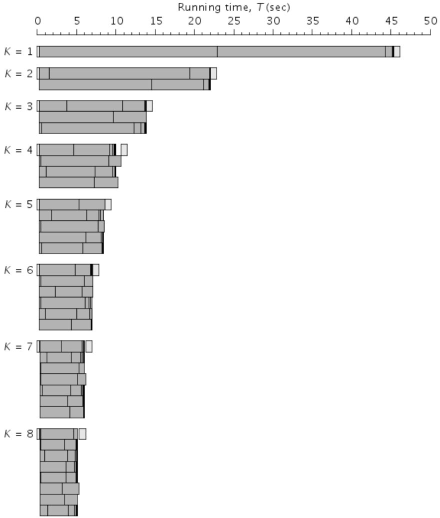
```
new IntegerForLoop() {
   public IntegerSchedule schedule() {
   /*
      return IntegerSchedule.fixed();     // fixed schedule
      return IntegerSchedule.dynamic();   // dynamic schedule; chunk sz = 1
      return IntegerSchedule.dynamic(5);  // dynamic schedule; chunk sz = 5
      return IntegerSchedule.guided();    // guided schedule; min chunk sz = 1
      return IntegerSchedule.guided(5);   // dynamic schedule; min chunk sz = 5
      return IntegerSchedule.runtime();   // runtime schedule; inherited method
   */
   }
   ...
}
```

- For a runtime schedule, set the `pj.schedule` property (default is `fixed`)

```
java -Dpj.schedule=guided ...

java -Dpj.schedule="guided(5)" ...
```

Guided Schedule Load Balance



Guided Schedule Load Balance

K	$T_p(K)$	$T_p(1)$	B
2	21778	45123	0.97
3	13607	45123	0.90
4	10401	45123	0.92
5	8332	45123	0.92
6	6796	45123	0.90
7	5926	45123	0.92
8	5103	45123	0.90

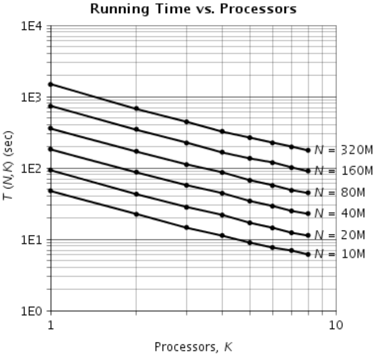
Guided Schedule Load Balance

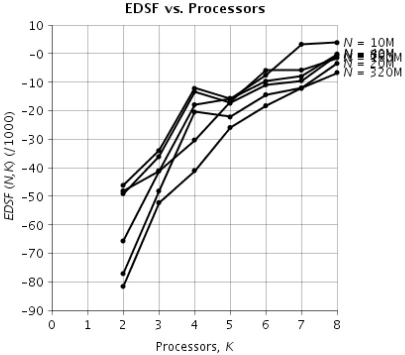
K	$T_p(K)$	$T_p(1)$	B
2	21778	45123	0.97
3	13607	45123	0.90
4	10401	45123	0.92
5	8332	45123	0.92
6	6796	45123	0.90
7	5926	45123	0.92
8	5103	45123	0.90

Recall:

- What is the "ideal" B?
- What is a "good" B?

Guided Schedule Running Time and EDSF

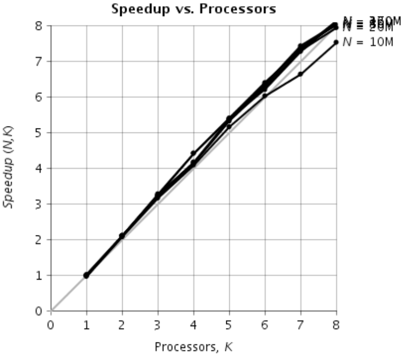


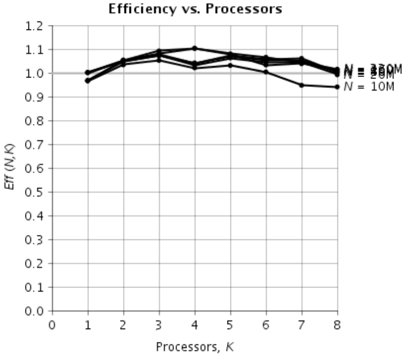


Recall:

- What does ***EDSF*** measure?

Guided Schedule Speedup and Efficiency



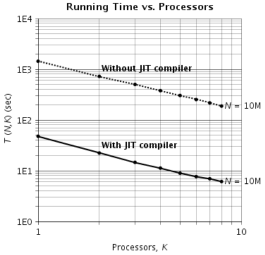


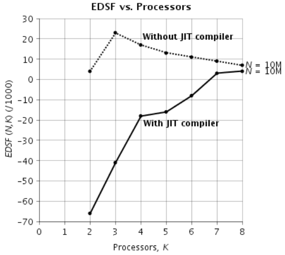
Recall:

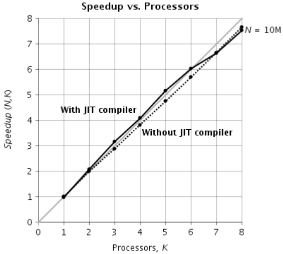
- What does ***Eff*** measure?

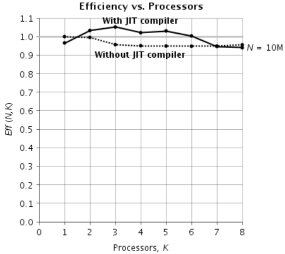
JIT Compiler Effect

Load balance (***B***) and efficiency (***Eff***) are "better than ideal".









JIT Compiler Effect

Load balance (***B***) and efficiency (***Eff***) are “better than ideal”.

Textbook claims:

- With a guided schedule and more threads, there are more chunks and more calls to the `IntegerForLoop.run(int first, int last)`. Therefore, JVM detects `run` as a hot-spot earlier, compiles to machine code sooner, and more of the program runs the faster machine code.
- The longer code runs, the more it is optimized. Fewer threads run longer, optimize more, run faster (compared the theoretical running time), appear to have greater speedup (compared to theoretical speedup), implying smaller sequential fraction. More threads run shorter, optimize less, run slower, appear to have less speedup, implying larger sequential fraction.

JIT Compiler Effect

Load balance (***B***) and efficiency (***Eff***) are "better than ideal".

Let's be critical:

- We were suspicious about the claim that the JIT compiler could explain the variation we saw in the AES Key Search program.
  - The `AES256Cipher.encrypt` method is called the $\mathbf{2^n}$ times, whether running sequentially or in parallel.
  - If JVM JIT compiles `AES256Cipher.encrypt` after **1000** invocations, then **1000** invocations are interpreted and $\mathbf{2^n - 1000}$ are compiled, again, whether running sequentially or in parallel.
- Does this reasoning apply to the Mandelbrot Set program?
  - Some number of pixels are calculated,
    whether running sequentially or in parallel.

JIT Compiler Effect

Load balance ($\boldsymbol{B}$) and efficiency ($\boldsymbol{Eff}$) are "better than ideal".

Let's be critical:

- We were suspicious about the claim that the JIT compiler could explain the variation we saw in the AES Key Search program.
  - The `AES256Cipher.encrypt` method is called the $\mathbf{2^n}$ times, whether running sequentially or in parallel.
  - If JVM JIT compiles `AES256Cipher.encrypt` after $\mathbf{1000}$ invocations, then $\mathbf{1000}$ invocations are interpreted and $\mathbf{2^n - 1000}$ are compiled, again, whether running sequentially or in parallel.
- Does this reasoning apply to the Mandelbrot Set program?
  - Some number of pixels are calculated,
    whether running sequentially or in parallel.
  - But, the sequential Mandelbrot Set program does not have a `run` method to be JIT compiled (assuming methods are the granularity of JIT compilation).

JIT Compiler Effect

Load balance (***B***) and efficiency (***Eff***) are "better than ideal".

Let's be critical:

- Does this reasoning apply to the Mandelbrot Set program?
  - Some number of pixels are calculated, whether running sequentially or in parallel.
  - But, the sequential Mandelbrot Set program does not have a `run` method to be JIT compiled (assuming methods are the granularity of JIT compilation).

I claim: This is a bad experiment; not comparing apples-to-apples.

- Speedup should be comparing against the best sequential program.
- But `MandelbrotSetSeq` isn't the best sequential program; it isn't getting the same benefit of JIT compilation.
- No "excuse" for super-linear speedups!

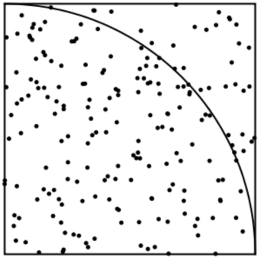
MandelbrotSetMethod{Seq,Smp}.java

```
code/MandelbrotSetMethodSeq.java
code/MandelbrotSetMethodSmp.java
```

Estimating π

Suppose we have a square dartboard with sides of length $\mathbf{1m}$ and with a quarter-circle of radius $\mathbf{1m}$ inscribed.

Now suppose we throw a large number of darts at the board. What fraction of the darts will land within the q-circle?



Estimating π

More formally:

- Let N be the number of darts thrown.
- Let C be the number of darts that landed within the q-circle.
- Then C/N should be approximately the same as the ratio of the q-circle's area to the square's area:

$$\frac{C}{N} \approx \frac{\frac{1}{4} \cdot \pi \cdot (1\mathrm{m})^2}{(1\mathrm{m})^2} = \frac{\pi}{4}$$

Estimating π

More formally:

- Let N be the number of darts thrown.
- Let C be the number of darts that landed within the q-circle.
- Then C/N should be approximately the same as the ratio of the q-circle's area to the square's area:

$$\frac{C}{N} \approx \frac{\frac{1}{4} \cdot \pi \cdot (1\mathrm{m})^2}{(1\mathrm{m})^2} = \frac{\pi}{4}$$

How can we use this method to compute the value of π?

Monte Carlo Algorithms

Algorithms that calculate their results using random numbers are called Monte Carlo algorithms.

Yes, we know much better ways of computing the value of π, but we don't know much better ways of computing:

- simulation of nuclear fusion and fission bombs (Manhattan Project)
- molecular dynamics
- weather forcasts (ensemble forecasting)
- global illumination for photorealistic images
- integrating complex formulas

Parallel Computing and Monte Carlo Algorithms

Why apply parallel computing to Monte Carlo algorithms?

Consider the algorithm for calculating the value of π:

- Monte Carlo algorithms yield an estimate.
- The estimate's accuracy is proportional to the square root of the number of random points.
  - If we want 10 times the accuracy (that is, an additional decimal place), then we need 100 times as many random points.

A reasonably accurate answer requires sampling many of random points. Thus, Monte Carlo algorithms are good candidates for parallel computing.

Pi{Seq,Smp}.java

code/PiSeq.java

Pi{Seq,Smp}.java

code/PiSeq.java

- What portion of the Pi program is parallelizable?
- Any concerns about load balance?
- Any concerns about the variables accessed in parallelized region?

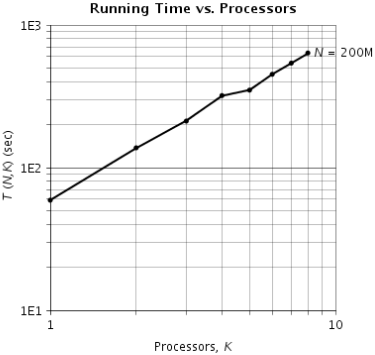
Pi{Seq,Smp}.java

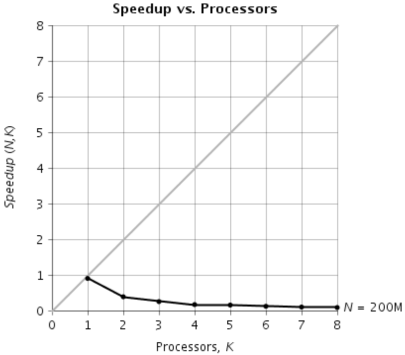
code/PiSeq.java

- What portion of the Pi program is parallelizable?
- Any concerns about load balance?
- Any concerns about the variables accessed in parallelized region?
  - Instances of `java.util.Random` are *multiple-thread safe*.
    - Multiple threads calling `prng.nextDouble()` will synchronize.
  - But `++ count` is a source of write-write conflicts.
    - `++ count` (equivalent to `count = count + 1`) is *not* atomic.
    - Use `java.util.concurrent.atomic.AtomicLong` or `edu.rit.pj.reduction.SharedLong`.

code/PiSmp.java

`PiSmp` Running Time and Speedup


Running Time vs. Processors


Speedup vs. Processors

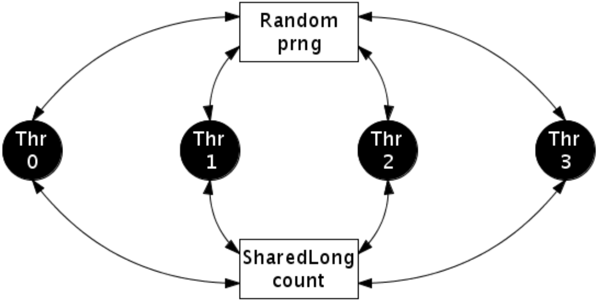
What has gone (horribly, horribly) wrong?
We're not just getting bad speedup, we're getting slowdown!

Reduction Pattern

Ironically, `PiSmp`'s multiple-thread safety is causing the slowdown!

Reduction Pattern

Ironically, `PiSmp`'s multiple-thread safety is causing the slowdown!



- Overhead due to synchronization outweighs the benefits of parallelism.
- (Also overhead due to cache coherency.)

Reduction Pattern

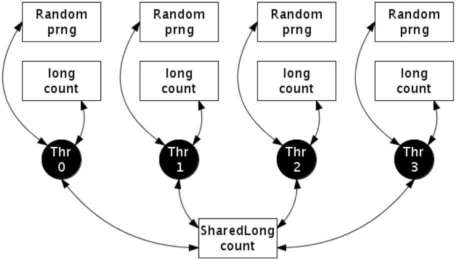
Solution:

- *per-thread* `prng_thread` and *per-thread* `count_thread`
- *shared* `count` only updated once per thread
  - (still synchronization overhead, but much less)

Reduction Pattern

Solution:

- *per-thread* `prng_thread` and *per-thread* `count_thread`
- *shared* `count` only updated once per thread
  - (still synchronization overhead, but much less)



- What should we be concerned about with per-thread variables?

PiSmp2.java

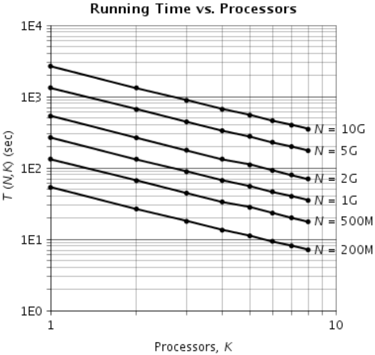
code/PiSmp2.java

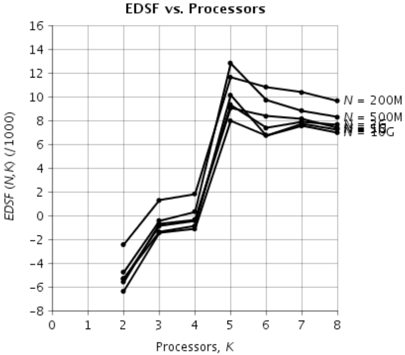
PiSmp2.java

code/PiSmp2.java

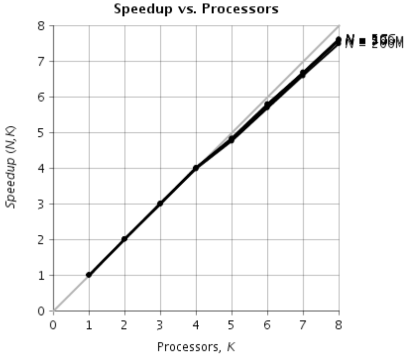
- Is there any *other* cache interference?
- Is there any *other* synchronization overhead?

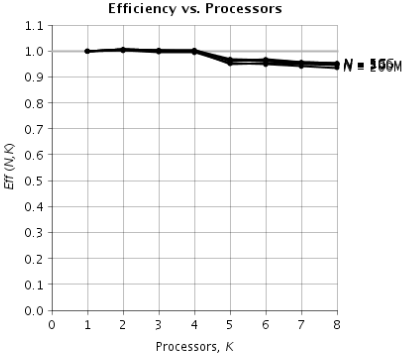
`PiSmp2` Running Time and EDSF





PiSmp2 Speedup and Efficiency





Did we actually parallelize `PiSeq`?

What is even more important than speedup or sizeup when we parallelize programs?

Did we actually parallelize `PiSeq`?

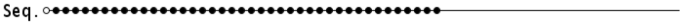
What is even more important than speedup or sizeup
when we parallelize programs?

Correctness! That our parallel program is the *same* program.
Let's check.

Did we actually parallelize PiSeq?

What is going on?

- Think about the sequence of random points generated by the sequential program.

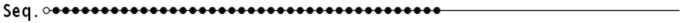


- Think about the sequences of random points generated by the parallel program.

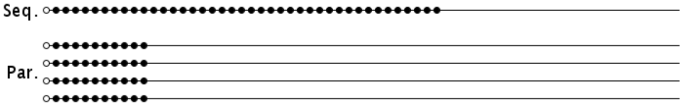
Did we actually parallelize PiSeq?

What is going on?

- Think about the sequence of random points generated by the sequential program.



- Think about the sequences of random points generated by the parallel program.



  - Each thread generates the *same* first $\boldsymbol{N/K}$ random points as is generated by the sequential program.
  - Using more processors *decreased* our accuracy!

So we have a different (and worse) program than we started with.
(Was PiSmp a correct parallelization of PiSeq?)

Parallel PRNG Patterns

We need a pseudorandom number generator (PRNG) that “plays nice” with parallelization.

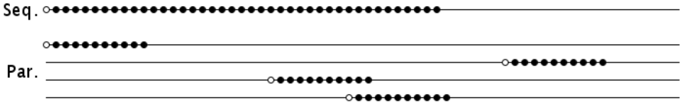
Alternatively, we need to use a PRNG in a way that “plays nice” with parallelization.

Parallel PRNG Patterns

We need a pseudorandom number generator (PRNG) that "plays nice" with parallelization.

Alternatively, we need to use a PRNG in a way that "plays nice" with parallelization.

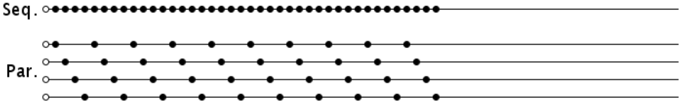
- Independent sequences
  - Each thread initializes its PRNG with a different seed.



Pros? Cons? Correct (w/ respect to sequential program and different K parallel programs)?

Parallel PRNG Patterns

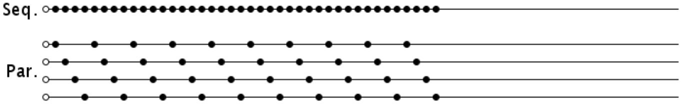
- Leapfrogging
  - Each thread leapfrogs (skips over) the PRNs of other threads.



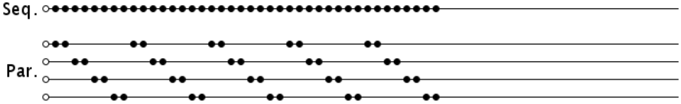
Pros? Cons? Correct (w/ respect to sequential program and different K parallel programs)?

Parallel PRNG Patterns

- Leapfrogging
  - Each thread leapfrogs (skips over) the PRNs of other threads.

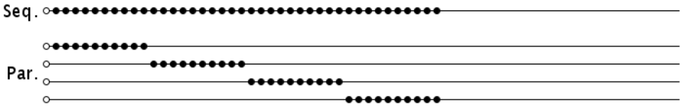


Pros? Cons? Correct (w/ respect to sequential program and different K parallel programs)?



Parallel PRNG Patterns

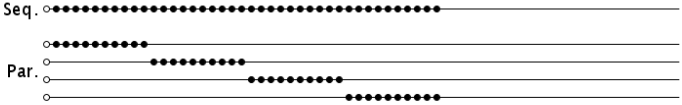
- Sequence splitting
  - Each thread initially skips $\boldsymbol{i} \cdot \boldsymbol{N}/\boldsymbol{K}$ PRNs (where $\boldsymbol{i}$ is the thread's number).



Pros? Cons? Correct (w/ respect to sequential program and different K parallel programs)?

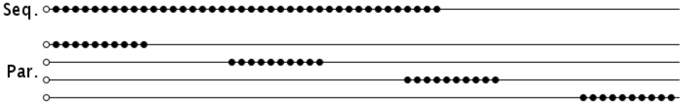
Parallel PRNG Patterns

- Sequence splitting
  - Each thread initially skips $\boldsymbol{i} \cdot \boldsymbol{N} / \boldsymbol{K}$ PRNs (where $\boldsymbol{i}$ is the thread's number).



Pros? Cons? Correct (w/ respect to sequential program and different K parallel programs)?

  - Each thread initially skips $\boldsymbol{i} \cdot \boldsymbol{M} / \boldsymbol{K}$ PRNs (where $\boldsymbol{i}$ is the thread's number and $\boldsymbol{M}$ is greater than any $\boldsymbol{N}$).



Pros? Cons? Correct (w/ respect to sequential program and different K parallel programs)?

Pseduorandom Number Generator Algorithms

A PRNG algorithm has two parts:

- Hash Function ($\boldsymbol{H(x)}$)
  - How to “randomize” an input ($\boldsymbol{x}$), yielding a random number.
- Mode of Operation
  - How to generate the sequence of inputs ($\boldsymbol{x}$s) for the hash function.

Pseudorandom Number Generator Algorithms

Hash functions

- Linear congruential generator (LCG)

$$H(x) = x \cdot a + b \pmod{m}$$

- Multiplicative congruential generator (MCG)

$$H(x) = x \cdot a \pmod{m}$$

- Xorshift generator (right-shift and left-shift)

```
$x \leftarrow x \oplus (x \gg a)$
$x \leftarrow x \oplus (x \ll b)$
$x \leftarrow x \oplus (x \gg c)$
return $x$
```

```
$x \leftarrow x \oplus (x \ll a)$
$x \leftarrow x \oplus (x \gg b)$
$x \leftarrow x \oplus (x \ll c)$
return $x$
```

- Composite hash function

$$H(x) = H_4(H_3(H_2(H_1(x))))$$

Pseudorandom Number Generator Algorithms

Mode of operation

- Iterated mode

```
$seed \leftarrow H(seed)$
return $seed$
```

- Counter mode

```
$seed \leftarrow seed + 1$
return $H(seed)$
```

A Parallel PRNG Class

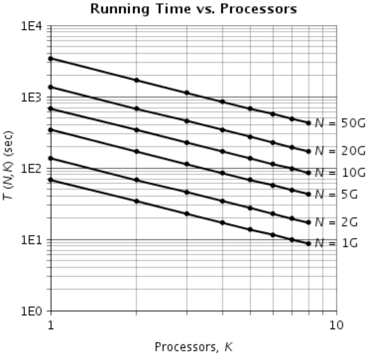
`edu.rit.util.Random`

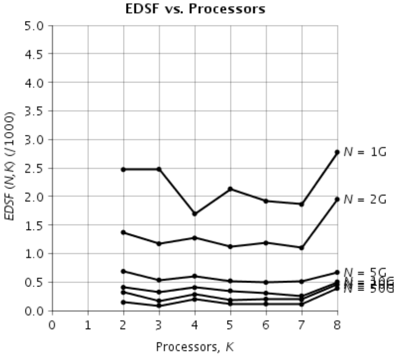
- Period
  - 2^{64} values before repeating sequence
- Skipping
  - `void skip()`
  - `void skip(long n)`
  - `int nextInt(long n)`
  - `double nextDouble(long n)`
- Not multiple-thread safe
  - avoid synchronization overhead

PiSmp3.java

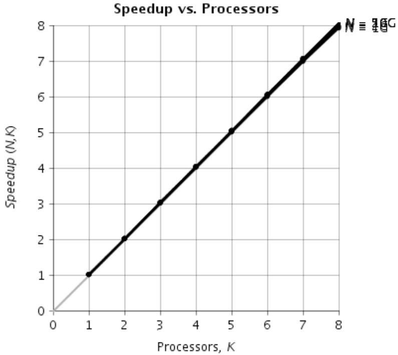
code/PiSmp3.java

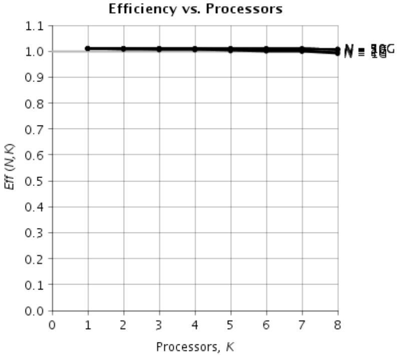
PiSmp3 Running Time and EDSF





PiSmp3 Speedup and Efficiency





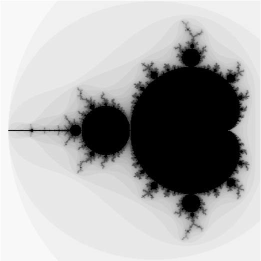
Histogram of the Mandelbrot Set

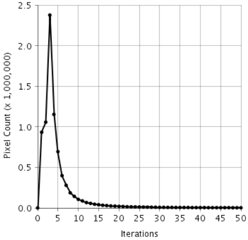
Compute the number of pixels whose iteration count was 0, 1, . . . , ***L***.

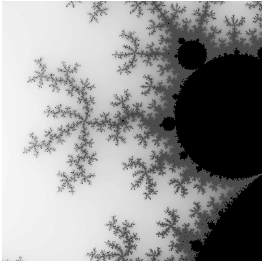
- Number of pixels whose iteration count was ***L*** estimates the area of the Mandelbrot set.

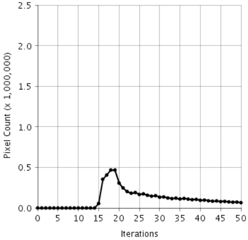
```
% java edu.rit.smp.fractal.MSHistogramSeq 3200 3200 -0.75 0 \
1200 1000 out.txt ; cat out.txt
0 msec pre
18629 msec calc
12 msec post
18641 msec total
0 0
1 933024
2 1056585
3 2376460
4 1151978
5 692974
...
995 6
996 2
997 0
998 6
999 14
1000 2175587
```

Histogram of the Mandelbrot Set







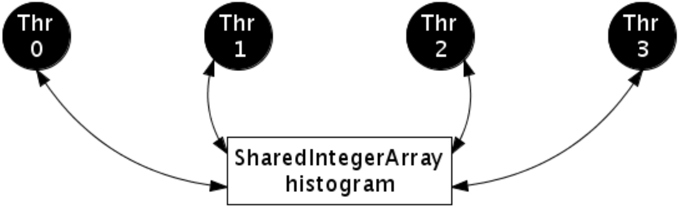


MSHistogram{Seq,Smp}.java

```
code/MSHistogramSeq.java
code/MSHistogramSmp.java
```

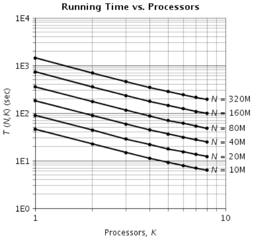
MSHistogram{Seq,Smp}.java

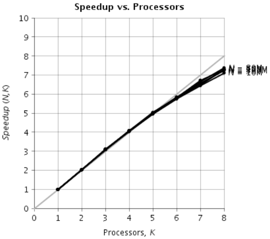
```
code/MSHistogramSeq.java
code/MSHistogramSmp.java
```

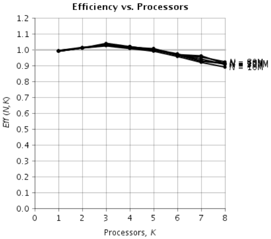


- Overhead due to synchronization outweighs the benefits of parallelism, just like `PiSmp`.

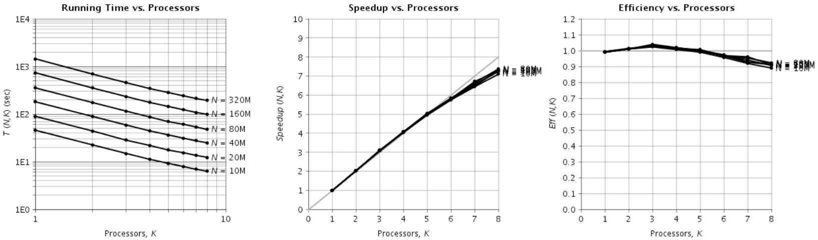
MSHistogramSmp Running Time, Speedup, and Eff





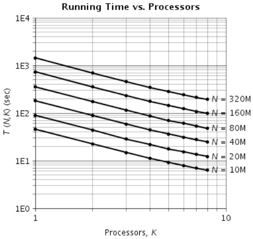


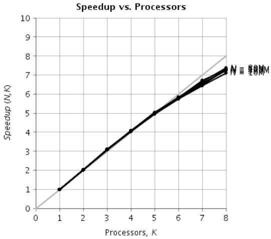
MSHistogramSmp Running Time, Speedup, and Eff

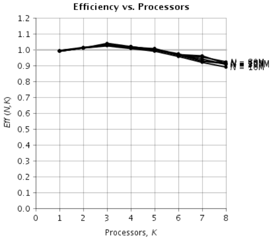


Huh? Why hasn't this gone (horribly, horribly) wrong?

`MSHistogramSmp` Running Time, Speedup, and Eff



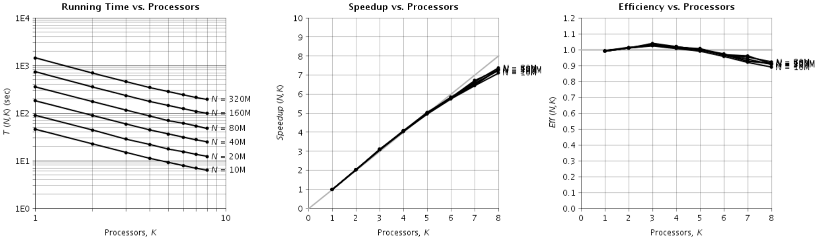




Huh? Why hasn't this gone (horribly, horribly) wrong?

Actually, overhead of (this kind of) synchronization is only high
when there high *contention* for a shared variable;
that is, when multiple thread access the variable at the same time.

MSHistogramSmp Running Time, Speedup, and Eff

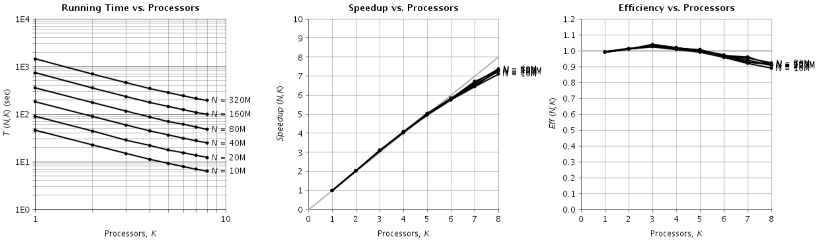


Huh? Why hasn't this gone (horribly, horribly) wrong?

Actually, overhead of (this kind of) synchronization is only high
when there high *contention* for a shared variable;
that is, when multiple thread access the variable at the same time.

And what do we think about super-linear speedups?

MSHistogramSmp Running Time, Speedup, and Eff



Huh? Why hasn't this gone (horribly, horribly) wrong?

Actually, overhead of (this kind of) synchronization is only high
when there high *contention* for a shared variable;
that is, when multiple thread access the variable at the same time.

And what do we think about super-linear speedups?

Nonetheless, can and should apply the reduction pattern.

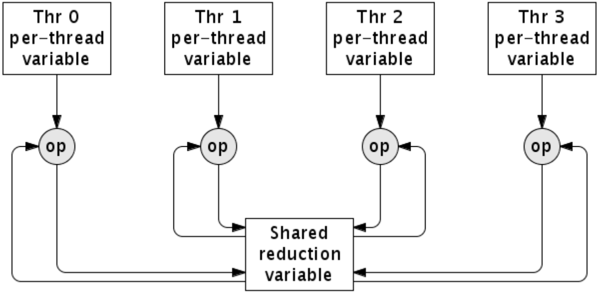
Reduction Operators

Elements of the parallel reduction pattern:

- a global shared reduction variable or variables;
  holds the complete result computed by whole program.
- per-thread variable or variables;
  hold the partial result computed by the thread.
- a computation to combine per-thread results
  into the global shared reduction variable or variables.

Reduction Operators

The computation to combine per-thread results
is often a reduction operator, a binary operation.



Thr 0 per–thread variable
Thr 1 per–thread variable
Thr 2 per–thread variable
Thr 3 per–thread variable
op
op
op
op
Shared reduction variable

Reduction operators must be commutative and associative.

- sum, product, maximum, minimum, and, or, xor

Reduction Operators

The edu.rit.reduction.IntegerOp class:

```
public abstract class IntegerOp {
  public abstract int op (int x, int y);

  public static final IntegerOp SUM = new IntegerOp () {
    public int op (int x, int y) {
      return x + y;
    }
  }
}
```

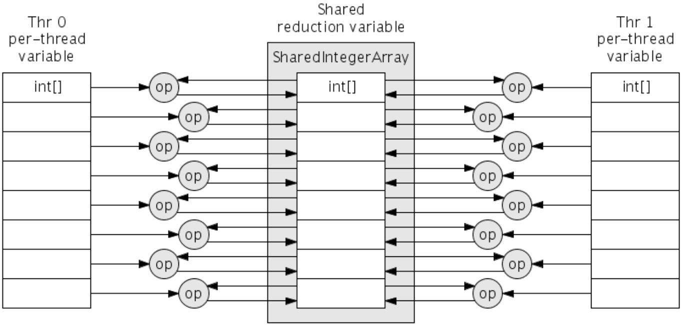
A custom reduction operator:

```
public class ModularSum extend IntegerOp {
  private int M;
  public ModularSum (int M) {
    this.M = M;
  }
  public int op (int x, int y) {
    return (x + y) % M;
  }
}
```

Reduction Operators

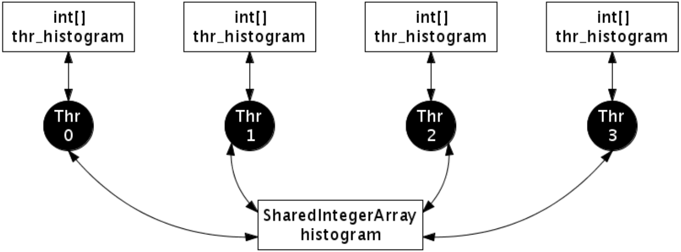
```
// SharedInteger class provides:
public int reduce(int value, IntegerOp op);

// SharedIntegerArray class provides:
public void reduce(int[] values, IntegerOp op);
```

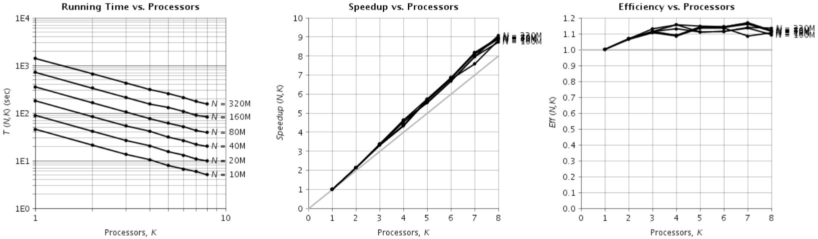


MSHistogramSmp2.java

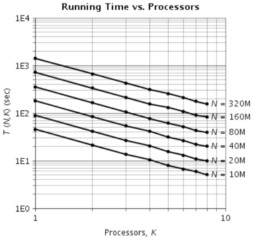
code/MSHistogramSmp2.java

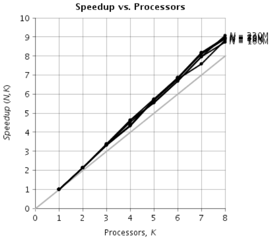


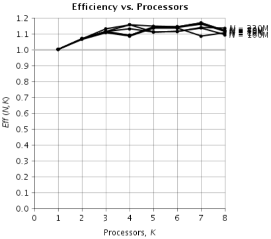
MSHistogramSmp2 Running Time, Speedup, and Eff



MSHistogramSmp2 Running Time, Speedup, and Eff







Bad experiment! No biscuit!

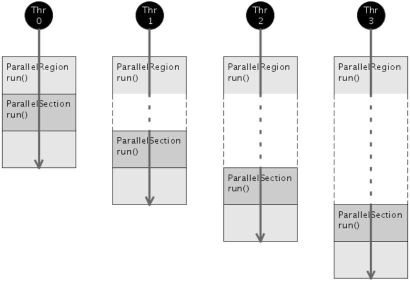
Critical Sections

If the reduction computation is more complicated than combining the reduction variable and the per-thread variables with a binary operation, then the program may require explicit thread synchronization.

Critical sections provide mutual exclusion to allow an arbitrary block of code to be run in a multiple-thread safe manner.

Critical Sections

```
new ParallelTeam().execute (new ParallelRegion() {
  public void run() {
    ...
    critical (new ParallelSection() {
      public void run() {
        // Code for the critical section
      }
    });
    ...
  }
});
```



MSHistogramSmp3.java

code/MSHistogramSmp3.java

MSHistogramSmp3 Running Time, Speedup, and Eff

Same as for MSHistogramSmp2.

MSHistogramSmp3 Running Time, Speedup, and Eff

Same as for MSHistogramSmp2.

What is the overhead tradeoff (for, say, 1000 iterations and 8 threads)?

- MSHistogramSmp2: SharedIntegerArray.reduce()

- MSHistogramSmp3: ParallelRegion.critical()

`MSHistogramSmp3` Running Time, Speedup, and Eff

Same as for `MSHistogramSmp2`.

What is the overhead tradeoff (for, say, 1000 iterations and 8 threads)?

- `MSHistogramSmp2`: `SharedIntegerArray.reduce()`
  - **1001** × **8** integer compare-and-swaps
- `MSHistogramSmp3`: `ParallelRegion.critical()`
  - **8** lock acquire-and-releases

MSHistogramSmp3 Running Time, Speedup, and Eff

Same as for MSHistogramSmp2.

What is the overhead tradeoff (for, say, 1000 iterations and 8 threads)?

- MSHistogramSmp2: SharedIntegerArray.reduce()
  - **1001** × **8** integer compare-and-swaps
- MSHistogramSmp3: ParallelRegion.critical()
  - **8** lock acquire-and-releases

Assuming a balanced load, all threads will perform reduction at same time.

`MSHistogramSmp3` Running Time, Speedup, and Eff

Same as for `MSHistogramSmp2`.

What is the overhead tradeoff (for, say, 1000 iterations and 8 threads)?

- `MSHistogramSmp2`: `SharedIntegerArray.reduce()`
  - **1001** × **8** integer compare-and-swaps
  - Parallelism in reduction (each thread updating a different index), but cache interference.
- `MSHistogramSmp3`: `ParallelRegion.critical()`
  - **8** lock acquire-and-releases
  - No parallelism in reduction,
    but cheap(er) operation in critical section.

Assuming a balanced load, all threads will perform reduction at same time.

`MSHistogramSmp3` Running Time, Speedup, and Eff

Same as for `MSHistogramSmp2`.

What is the overhead tradeoff (for, say, 1000 iterations and 8 threads)?

- `MSHistogramSmp2`: `SharedIntegerArray.reduce()`
  - **1001** × **8** integer compare-and-swaps
  - Parallelism in reduction (each thread updating a different index), but cache interference.
- `MSHistogramSmp3`: `ParallelRegion.critical()`
  - **8** lock acquire-and-releases
  - No parallelism in reduction,
    but cheap(er) operation in critical section.

Assuming a balanced load, all threads will perform reduction at same time.

Finally, compared to `MSHistogramSmp`, we gained time but lost . . .

`MSHistogramSmp3` Running Time, Speedup, and Eff

Same as for `MSHistogramSmp2`.

What is the overhead tradeoff (for, say, 1000 iterations and 8 threads)?

- `MSHistogramSmp2`: `SharedIntegerArray.reduce()`
  - **1001** × **8** integer compare-and-swaps
  - Parallelism in reduction (each thread updating a different index), but cache interference.
- `MSHistogramSmp3`: `ParallelRegion.critical()`
  - **8** lock acquire-and-releases
  - No parallelism in reduction,
    but cheap(er) operation in critical section.

Assuming a balanced load, all threads will perform reduction at same time.

Finally, compared to `MSHistogramSmp`, we gained time but lost space.

Combining Partial Results

- No reduction pattern; just update the shared variable using a multiple-thread-safe operation.
- Reduction pattern; each thread accumulates its partial result in a per-thread variable, then, as its last act, updates the shared variable (once) using a multiple-thread-safe operation.
- Reduction pattern with reduction operator; each thread accumulates its partial result in a per-thread variable, then, as its last act, updates the shared variable (once) using a reduction operation (just a special, but common, case of the previous).
- Reduction pattern with critical section; each thread accumulates its partial result in a per-thread variable, then, as its last act, updates the shared variable (once) inside a critical section.

Last three are really all the same, just different emphasis on
how well-behaved is the update operation and
who is responsible for multiple-thread safety.