Administration

- Assignment #2 Posted
  - Due January 17 (Mon.)

- Midterm exam
  - January 11 (Tues.)
  - Open book, Open notes
  - Will need "calculator"
    - May use laptop, iPhone, etc.

Parallel Computing I

SMP: Sequential Dependencies, Barrier Actions, and Overlapping

Looking Back, Looking Forward

Last three weeks:

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

This week:

- sequential dependencies
- barrier actions
- overlapping

Quick Review

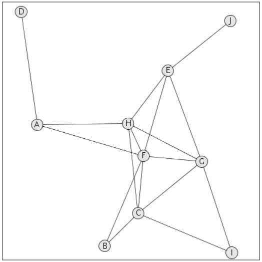
- `FindKey`
- `MandelbrotSet`
- `Pi`
- `MSHistogram`

Performance on `cadmium`

- Four AMD Opteron 6172 12-core CPUs (48 processors), 2.1 GHz clock, 128 GB main memory

All-Pairs Shortest-Path Problem

Given a graph with weighted edges,
determine the (length of the) shortest path between all pairs of vertices.



	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
A	0	∞	∞	462	∞	451	∞	370	∞	∞
B	∞	0	190	∞	∞	399	∞	∞	∞	∞
C	∞	190	0	∞	∞	234	333	366	414	∞
D	462	∞	∞	0	∞	∞	∞	∞	∞	∞
E	∞	∞	∞	∞	0	359	394	269	∞	325
F	451	399	234	∞	359	0	239	144	∞	∞
G	∞	∞	333	∞	394	239	0	337	389	∞
H	370	∞	366	∞	269	144	337	0	∞	∞
I	∞	∞	414	∞	∞	∞	389	∞	0	∞
J	∞	∞	∞	∞	325	∞	∞	∞	∞	0

Floyd's All-Pairs Shortest-Path Algorithm

for $i = 0$ to $n - 1$
    for $r = 0$ to $n - 1$
        for $c = 0$ to $n - 1$
            // Update the distance from r *to* c *via* i*.*
            $d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})$

Floyd's All-Pairs Shortest-Path Algorithm

for $i = 0$ to $n - 1$
    for $r = 0$ to $n - 1$
        for $c = 0$ to $n - 1$
            // Update the distance from r *to* c *via* i.
            $d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})$

	A	B	C	D	E	F	G	H	I	J
A	0	∞	∞	462	∞	451	∞	370	∞	∞
B	∞	0	190	∞	∞	399	∞	∞	∞	∞
C	∞	190	0	∞	∞	234	333	366	414	∞
D	462	∞	∞	0	∞	∞	∞	∞	∞	∞
E	∞	∞	∞	∞	0	359	394	269	∞	325
F	451	399	234	∞	359	0	239	144	∞	∞
G	∞	∞	333	∞	394	239	0	337	389	∞
H	370	∞	366	∞	269	144	337	0	∞	∞
I	∞	∞	414	∞	∞	∞	389	∞	0	∞
J	∞	∞	∞	∞	325	∞	∞	∞	∞	0

Which entry is the first to be changed?

Floyd's All-Pairs Shortest-Path Algorithm

for $i = 0$ to $n - 1$
    for $r = 0$ to $n - 1$
        for $c = 0$ to $n - 1$
            // Update the distance from r *to* c *via* i.
            $d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})$

	A	B	C	D	E	F	G	H	I	J
A	0	∞	∞	462	∞	451	∞	370	∞	∞
B	∞	0	190	∞	∞	399	∞	∞	∞	∞
C	∞	190	0	∞	∞	234	333	366	414	∞
D	462	∞	∞	0	∞	∞	∞	∞	∞	∞
E	∞	∞	∞	∞	0	359	394	269	∞	325
F	451	399	234	∞	359	0	239	144	∞	∞
G	∞	∞	333	∞	394	239	0	337	389	∞
H	370	∞	366	∞	269	144	337	0	∞	∞
I	∞	∞	414	∞	∞	∞	389	∞	0	∞
J	∞	∞	∞	∞	325	∞	∞	∞	∞	0

Which entry is the first to be changed?

Floyd's All-Pairs Shortest-Path Algorithm

for $i = 0$ to $n-1$
  for $r = 0$ to $n-1$
    for $c = 0$ to $n-1$
      // Update the distance from r *to* c *via* i.
      $d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})$

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
A	0	∞	∞	462	∞	451	∞	370	∞	∞
B	∞	0	190	∞	∞	399	∞	∞	∞	∞
C	∞	190	0	∞	∞	234	333	366	414	∞
D	462	∞	∞	0	∞	∞	∞	∞	∞	∞
E	∞	∞	∞	∞	0	359	394	269	∞	325
F	451	399	234	∞	359	0	239	144	∞	∞
G	∞	∞	333	∞	394	239	0	337	389	∞
H	370	∞	366	∞	269	144	337	0	∞	∞
I	∞	∞	414	∞	∞	∞	389	∞	0	∞
J	∞	∞	∞	∞	325	∞	∞	∞	∞	0

Which entry is the first to be changed?

Floyd's All-Pairs Shortest-Path Algorithm

for $i = 0$ to $n-1$
    for $r = 0$ to $n-1$
        for $c = 0$ to $n-1$
            // Update the distance from r *to* c *via* i.
            $d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})$

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
A	0	∞	∞	462	∞	451	∞	370	∞	∞
B	∞	0	190	∞	∞	399	∞	∞	∞	∞
C	∞	190	0	∞	∞	234	333	366	414	∞
D	462	∞	∞	0	∞	913	∞	∞	∞	∞
E	∞	∞	∞	∞	0	359	394	269	∞	325
F	451	399	234	∞	359	0	239	144	∞	∞
G	∞	∞	333	∞	394	239	0	337	389	∞
H	370	∞	366	∞	269	144	337	0	∞	∞
I	∞	∞	414	∞	∞	∞	389	∞	0	∞
J	∞	∞	∞	∞	325	∞	∞	∞	∞	0

Which entry is the first to be changed?

Floyd's All-Pairs Shortest-Path Algorithm

for $i = 0$ to $n - 1$
  for $r = 0$ to $n - 1$
    for $c = 0$ to $n - 1$
      // *Update the distance from* r *to* c *via* i.
      $d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})$

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
A	0	850	685	462	639	451	690	370	1079	964
B	850	0	190	1312	758	399	523	543	604	1083
C	685	190	0	1147	593	234	333	366	414	918
D	462	1312	1147	0	1101	913	1152	832	1541	1426
E	639	758	593	1101	0	359	394	269	783	325
F	451	399	234	913	359	0	239	144	628	684
G	690	523	333	1152	394	239	0	337	389	719
H	370	543	366	832	269	144	337	0	726	594
I	1079	604	414	1541	783	628	389	726	0	1108
J	964	1083	918	1426	325	684	719	594	1108	0

In general, original entries may change
and distance matrix may not be symmetric.

Input/Output Files

Represent distance matrix as an instance of `edu.rit.io.DoubleMatrixFile`:

- Input

```
DoubleMatrixFile dmf = new DoubleMatrixFile();
DoubleMatridFile.Reader reader = dmf.prepareToRead (instream);
reader.read();
reader.close();
int R = dmf.getRowCount();
int C = dmf.getColCount();
double[][] matrix = dmf.getMatrix();
```

- Output

```
double[][] matrix = new double [R] [C];
DoubleMatrixFile dmf = new DoubleMatrixFile( R, C, matrix);
DoubleMatridFile.Writer writer = dmf.prepareToWrite (outstream);
writer.write();
writer.close();
```

`FloydRandom.java` and `FloydSeq.java`

`code/FloydRandom.java`
`code/FloydSeq.java`

FloydRandom.java and FloydSeq.java

code/FloydRandom.java
code/FloydSeq.java

```
for (int i = 0; i < n; ++i) { double[] d_i = d[i];
  for (int r = 0; r < n; ++r) { double[] d_r = d[r];
    for (int c = 0; c < n; ++c) {
      d_r[c] = Math.min (d_r[c], d_r[i] + d_i[c]);  } } }
```

vs.

```
for (int i = 0; i < n; ++i) {
  for (int r = 0; r < n; ++r) {
    for (int c = 0; c < n; ++c) {
      d[r][c] = Math.min (d[r][c], d[r][i] + d[i][c]);  } } }
```

Parallelizing Floyd's Algorithm

How do we convert the sequential Floyd program
to a parallel Floyd program?

- What portions of the Floyd program are not parallelizable?
- What portions of the Floyd program might be parallelizable?

Parallelizing Floyd's Algorithm

Which loops are parallelizable?

$$d_{rc} \;\leftarrow\; \min(d_{rc}, d_{ri} + d_{ic})$$

- `for (int i = 0; i < n; ++i) { ... }`
  - On each iteration, store a value into every d_{rc} that depends upon the values of d_{rc}, d_{ri}, and d_{ic}, any of which could have been changed on the *previous* iteration.
  - There is a *sequential dependency* from each iteration i to the next.
  - This loop cannot be parallelized.

Parallelizing Floyd's Algorithm

Which loops are parallelizable?

$$d_{rc} \;\leftarrow\; \min(d_{rc}, d_{ri} + d_{ic})$$

- `for (int r = 0; r < n; ++r) { ... }`
  - On each iteration, store a value into every d_{rc} that depends upon the values of d_{rc}, d_{ri}, and d_{ic}.
  - Updates to d_{ic} (when $r = i$) will affect updates to d_{rc} (when $r > i$).

Parallelizing Floyd's Algorithm

Which loops are parallelizable?

$$d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})$$

- `for (int r = 0; r < n; ++r) { ... }`
  - On each iteration, store a value into every d_{rc} that depends upon the values of d_{rc}, d_{ri}, and d_{ic}.
  - Updates to d_{ic} (when $r = i$) will affect updates to d_{rc} (when $r > i$).
  - But, when $r = i$: $d_{ri} \leftarrow \min(d_{ic}, d_{ii} + d_{ic})$
  - Assuming $d_{ii} = 0$, the updates to d_{ic} (when $r = i$) are idempotent.
  - This loop can be parallelized.

Parallelizing Floyd's Algorithm

Which loops are parallelizable?

$$d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})$$

- `for (int r = 0; r < n; ++r) { ... }`
  - On each iteration, store a value into every d_{rc} that depends upon the values of d_{rc}, d_{ri}, and d_{ic}.
  - Updates to d_{ic} (when $r = i$) will affect updates to d_{rc} (when $r > i$).
  - But, when $r = i$: $d_{ri} \leftarrow \min(d_{ic}, d_{ii} + d_{ic})$
  - Assuming $d_{ii} = 0$, the updates to d_{ic} (when $r = i$) are idempotent.
  - This loop can be parallelized.

  - Any synchronization issues?

Parallelizing Floyd's Algorithm

Which loops are parallelizable?

$$d_{rc} \;\leftarrow\; \min(d_{rc}, d_{ri} + d_{ic})$$

- `for (int r = 0; r < n; ++r) { ... }`
  - On each iteration, store a value into every d_{rc} that depends upon the values of d_{rc}, d_{ri}, and d_{ic}.
  - Updates to d_{ic} (when $r = i$) will affect updates to d_{rc} (when $r > i$).
  - But, when $r = i$: $d_{ri} \;\leftarrow\; \min(d_{ic}, d_{ii} + d_{ic})$
  - Assuming $d_{ii} = 0$, the updates to d_{ic} (when $r = i$) are idempotent.
  - This loop can be parallelized.
  - Any synchronization issues?
    - Jave does not guarantee atomicity of reads and writes of a **`double`**.

Parallelizing Floyd's Algorithm

Which loops are parallelizable?

$$d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})$$

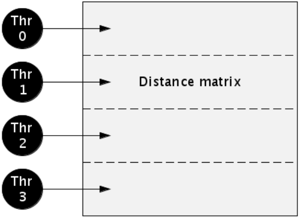
- `for (int r = 0; r < n; ++r) { ... }`
  - On each iteration, store a value into every d_{rc} that depends upon the values of d_{rc}, d_{ri}, and d_{ic}.
  - Updates to d_{ic} (when $r = i$) will affect updates to d_{rc} (when $r > i$).
  - But, when $r = i$: $d_{ri} \leftarrow \min(d_{ic}, d_{ii} + d_{ic})$
  - Assuming $d_{ii} = 0$, the updates to d_{ic} (when $r = i$) are idempotent.
  - This loop can be parallelized.
  - Any synchronization issues?
    - Jave does not guarantee atomicity of reads and writes of a **`double`**.
    - "For the purposes of the Java programming language memory model, a single write to a non-volatile long or double value is treated as two separate writes: one to each 32-bit half. This can result in a situation where a thread sees the first 32 bits of a 64 bit value from one write, and the second 32 bits from another write."

Parallelizing Floyd's Algorithm

Which loops are parallelizable?

$$d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})$$

- `for (int r = 0; r < n; ++r) { ... }`
  - This loop can be parallelized.



Row slicing

for $i = 0$ to $n - 1$
    pfor $r = 0$ to $n - 1$
        for $c = 0$ to $n - 1$
            $d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})$

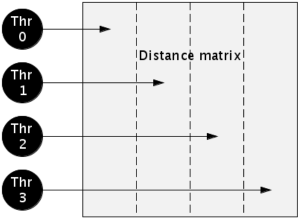
  - Any load-balancing issues?

Parallelizing Floyd's Algorithm

Which loops are parallelizable?

$$d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})$$

- `for (int c = 0; c < n; ++c) { ... }`
  - This loop can be parallelized. (Same analysis as before.)



Column slicing

for $i = 0$ to $n-1$
  for $r = 0$ to $n-1$
    pfor $c = 0$ to $n-1$
      $d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})$

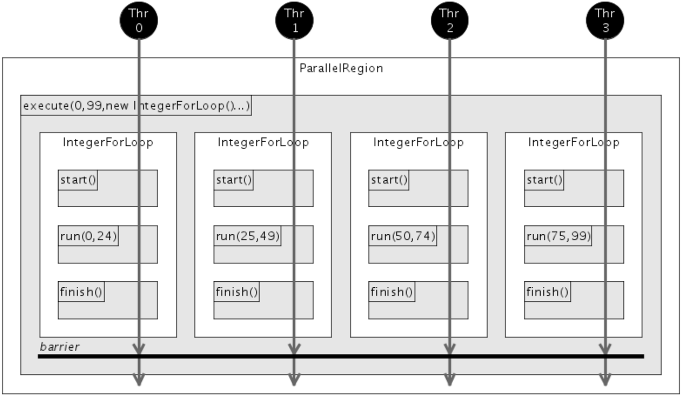
  - Any load-balancing issues?

FloydSmpRow.java

code/FloydSmpRow.java

FloydSmpRow.java

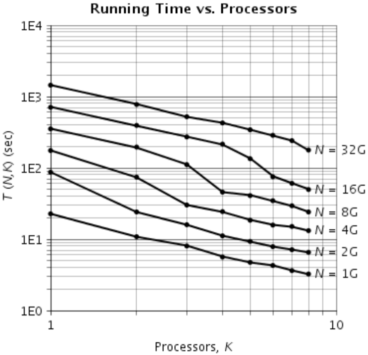
code/FloydSmpRow.java

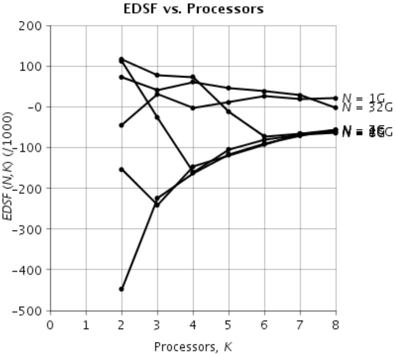


FloydSmpAltRow.java

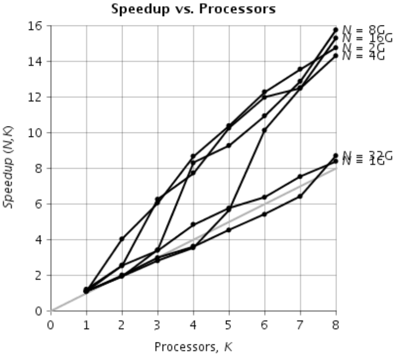
code/FloydSmpAltRow.java

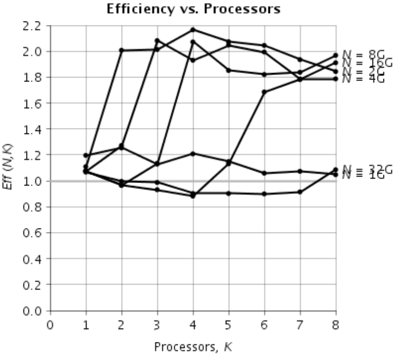
FloydSmpRow Running Time and EDSF



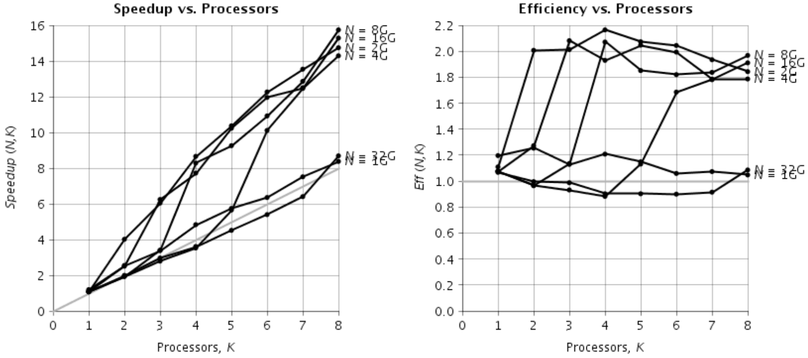


FloydSmpRow Speedup and Efficiency





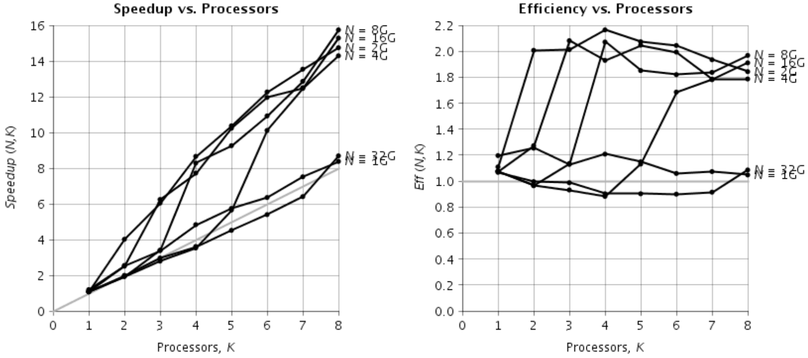
FloydSmpRow Speedup and Efficiency



The "unfair" JIT-compiler effect:

- `FloydSmpRow` has a "hot" `IntegerForLoop.run()` method.
- `FloydSeq` has a "naked" **`for`**-loop.

FloydSmpRow Speedup and Efficiency



The "unfair" JIT-compiler effect:

- FloydSmpRow has a "hot" IntegerForLoop.run() method.
- FloydSeq has a "naked" **for**-loop.

More than "just" the JIT-compiler effect.

`FloydSmpRow` Results

An abrupt jump in efficiencies as K increases.

- Larger N requires greater K before jump.

$$
\begin{array}{l}
\textbf{for } i = 0 \textbf{ to } n-1 \\
\quad \textbf{pfor } r = 0 \textbf{ to } n-1 \\
\quad\quad \textbf{for } c = 0 \textbf{ to } n-1 \\
\quad\quad\quad d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})
\end{array}
$$

The distance matrix requires $\approx 8n^2$ bytes.

Each thread accesses only $\approx 8(\frac{n^2}{K} + n)$ bytes per i iteration.

Futhermore, each thread accesses the *same* $\approx 8\frac{n^2}{K}$ bytes each i iteration.

`FloydSmpRow` Results

An abrupt jump in efficiencies as K increases.

- Larger N requires greater K before jump.

$$
\begin{array}{l}
\text{for } i = 0 \text{ to } n-1 \\
\quad \text{pfor } r = 0 \text{ to } n-1 \\
\quad\quad \text{for } c = 0 \text{ to } n-1 \\
\quad\quad\quad d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})
\end{array}
$$

The distance matrix requires $\approx 8n^2$ bytes.

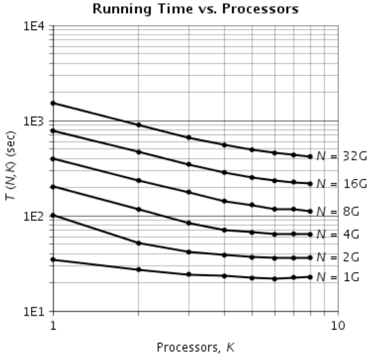
Each thread accesses only $\approx 8(\frac{n^2}{K} + n)$ bytes per i iteration.
Futhermore, each thread accesses the *same* $\approx 8\frac{n^2}{K}$ bytes each i iteration.

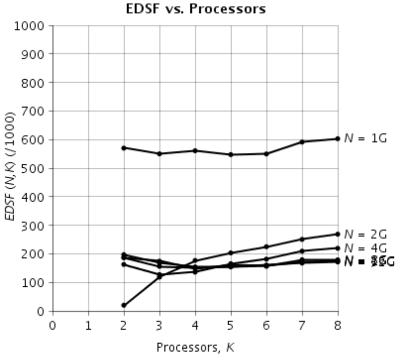
When threads' data all fits in their processors' cache,
parallel program performs *much* better than sequential,
which suffers from continual cache churning.

FloydSmpCol.java

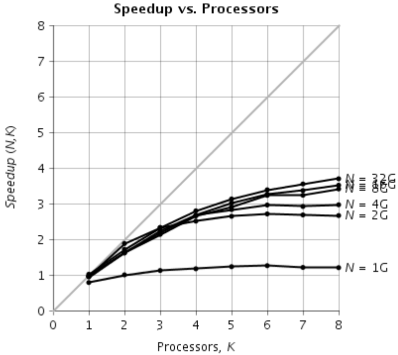
code/FloydSmpCol.java

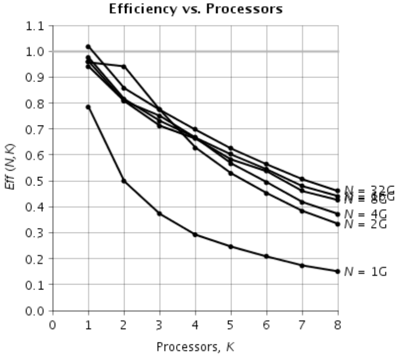
`FloydSmpColSmp` Running Time and EDSF





`FloydSmpColSmp` Speedup and Efficiency





`FloydSmpCol` Results

Classic Amdahl's Law behavior.

- Speedups approaching a limit
- Efficiencies continually decreasing as K increases
- Constant sequential fraction

In fact, a very large sequential fraction.

Why does `FloydSmpCol` have a larger sequential fraction?

`FloydSmpCol` Results

Classic Amdahl's Law behavior.

- Speedups approaching a limit
- Efficiencies continually decreasing as K increases
- Constant sequential fraction

In fact, a very large sequential fraction.

Why does `FloydSmpCol` have a larger sequential fraction?

`FloydSmpCol` requires n^2 barrier waits,
but `FloydSmpRow` requires only n barrier waits.

Parallelizing the innermost loop typical yields poor performance.

FloydSimpleRev{Seq,SmpCol}Method.java

What if we reverse the loops?

code/FloydSimpleSeqMethod.java
code/FloydSimpleRevSeqMethod.java
code/FloydSimpleRevSmpColMethod.java

Now, only n barrier waits.

As before, each thread accesses only $\approx 8(\frac{n^2}{K} + n)$ bytes per i iteration and each thread accesses the *same* $\approx 8\frac{n^2}{K}$ bytes each i iteration.

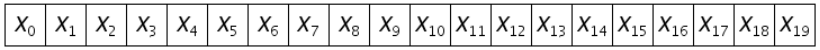
Same results?

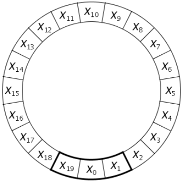
Cellular Automata

A cellular automaton (CA) is a simple abstract computing device that is capable of generating many kinds of interesting behavior.

The state of the system consists of a regular grid of cells (each of which has a value). At each time step, each cell simultaneously changes its value based on the values of a neighborhood of cells (according to some fixed rule).

A one-dimensional cellular automaton (1-D CA) uses an array of cells and the neighborhood of a cell consists of the cell to the left, the cell itself, and the cell to the right (using wraparound boundaries).





Elementary Cellular Automata

An elementary cellular automaton (ECA)
is a one-dimensional discrete cellular automaton (1-D DCA)
where each cell has a value that is either `0` or `1`.

The rule for updating cells can be represented as an unsigned **8**-bit integer

- interpret the left, center, and right cells as an unsigned **3**-bit integer called $\boldsymbol{n}$ and the new state of the center cell is the $\boldsymbol{n}^{\mathrm{th}}$ bit of the **8**-bit rule.

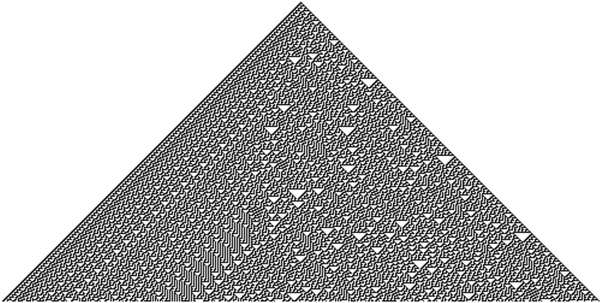
For instance, rule $\mathbf{30} = \mathbf{00011110_2}$ corresponds to the following:

configuration of cells	**111**	**110**	**101**	**100**	**011**	**010**	**001**	**000**
new state of center cell	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**0**

Elementary Cellular Automata

For instance, rule $\mathbf{30} = \mathbf{00011110_2}$ corresponds to the following:

configuration of cells	**111**	**110**	**101**	**100**	**011**	**010**	**001**	**000**
new state of center cell	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**0**



500 cells, 250 steps, rule $\mathbf{30} = \mathbf{00011110_2}$

ElementaryCA{Seq,SmpAlt}.java

```
boolean[][] cells = new boolean[S+1][N];
cells[0][c/2] = true;
for (int s = 1; s <= S; s++)
  for (int c = 0; c < N; c++)
    cells[s][c] = applyRule(cells[s-1][c-1],
                            cells[s-1][c],
                            cells[s-1][c+1]);
int total = 0;
for (int c = 0; c < N; c++)
  if (cells[S][c]) total++;
```

Requires $O(SN)$ memory to hold data;
problematic when scaling up the problem.

ElementaryCA{Seq,SmpAlt}.java

Only need previous row to calculate next row.

```
boolean[] next = new boolean[N];
boolean[] cells = new boolean[N];
cells[c/2] = true;
for (int s = 1; s <= S; s++) {
  for (int c = 0; c < N; c++)
    next[c] = applyRule(cells[c-1], cells[c], cells[c+1]);
  boolean[] temp = cells;
  cells = next;
  next = temp;
}
for (int c = 0; c < N; c++)
  if (cells[c]) total++;
```

Requires only $O(N)$ memory to hold data.

ElementaryCA{Seq,SmpAlt}.java

`code/ElementaryCASeq.java`
`code/ElementaryCASmpAlt.java`

Parallelizing `ElementaryCA`:

- What portions of the Elementary CA program are not parallelizable?
- What portions of the Elementary CA program are parallelizable?
  - What parallelization patterns?
- Any synchronization issues?
- Any load-balancing issues?
- Any cache interference?
- Any beneficial cache effects?

Continuous Cellular Automata

Another CA is a one-dimensional continuous cellular automaton (1-D CCA) where each cell has a value that is a rational number in the range $\mathbf{0}$ to $\mathbf{1}$.

The rule for updating cells uses two rational constants A and B:

$$x_i^{\mathrm{new}} = \mathrm{frac}\left(\frac{x_{i-1} + x_i + x_{i+1}}{3} \cdot A + B\right)$$

Continuous Cellular Automata

$$x_i^{\mathrm{new}} = \mathrm{frac}\left(\frac{X_{i-1} + X_i + X_{i+1}}{3} \cdot A + B\right); A = 1, B = 11/12$$

s	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
0	0	0	0	0	0	1	0	0	0	0

Continuous Cellular Automata

$$x_i^{\mathrm{new}} = \mathrm{frac}\left(\frac{X_{i-1} + X_i + X_{i+1}}{3} \cdot A + B\right); A = 1, B = 11/12$$

s	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
0	0	0	0	0	0	1	0	0	0	0
1	11/12	11/12	11/12	11/12	1/4	1/4	1/4	11/12	11/12	11/12

Continuous Cellular Automata

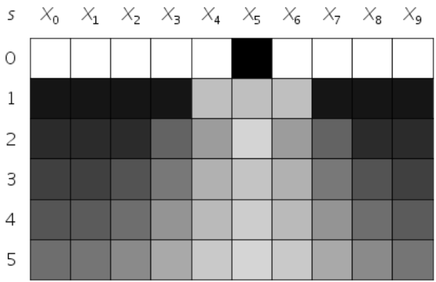
$$x_i^{\text{new}} = \text{frac}\left(\frac{X_{i-1} + X_i + X_{i+1}}{3} \cdot A + B\right); A = 1, B = 11/12$$

s	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
0	0	0	0	0	0	1	0	0	0	0
1	11/12	11/12	11/12	11/12	1/4	1/4	1/4	11/12	11/12	11/12
2	5/6	5/6	5/6	11/18	7/18	1/6	7/18	11/18	5/6	5/6
3	3/4	3/4	73/108	19/36	11/36	25/108	11/36	19/36	73/108	3/4
4	2/3	52/81	46/81	34/81	22/81	16/81	22/81	34/81	46/81	52/81
5	$\frac{551}{972}$	$\frac{527}{972}$	$\frac{149}{324}$	$\frac{109}{324}$	$\frac{23}{108}$	$\frac{53}{324}$	$\frac{23}{108}$	$\frac{109}{324}$	$\frac{149}{324}$	$\frac{527}{972}$

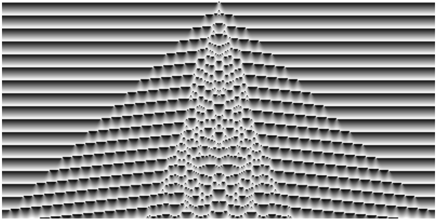
Continuous Cellular Automata

$$x_i^{\mathrm{new}} = \mathrm{frac}\left(\frac{X_{i-1} + X_i + X_{i+1}}{3} \cdot A + B\right); A = 1, B = 11/12$$

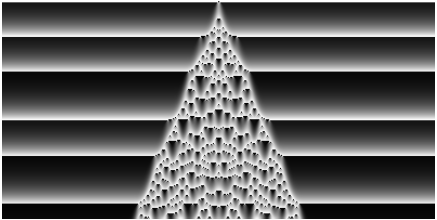
s	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
0	0	0	0	0	0	1	0	0	0	0
1	11/12	11/12	11/12	11/12	1/4	1/4	1/4	11/12	11/12	11/12
2	5/6	5/6	5/6	11/18	7/18	1/6	7/18	11/18	5/6	5/6
3	3/4	3/4	73/108	19/36	11/36	25/108	11/36	19/36	73/108	3/4
4	2/3	52/81	46/81	34/81	22/81	16/81	22/81	34/81	46/81	52/81
5	$\frac{551}{972}$	$\frac{527}{972}$	$\frac{149}{324}$	$\frac{109}{324}$	$\frac{23}{108}$	$\frac{53}{324}$	$\frac{23}{108}$	$\frac{109}{324}$	$\frac{149}{324}$	$\frac{527}{972}$



Continuous Cellular Automata



400 cells, 200 steps, $A = 1$, $B = 11/12$



400 cells, 200 steps, $A = 13/12$, $B = 11/12$

Continuous Cellular Automata

Use rational arithmetic, not floating-point arithmetic.

- Floating-point arithmetic does not have sufficient precision; rounding errors would quickly accumulate and lead to incorrect results.
- `edu.rit.numeric.BigRational`
  - Represent numerator and denominator with arbitrary precision integers (`java.math.BigInteger`).
  - Convert to **`float`** or **`double`** via arbitrary precision decimals (`java.math.BigDecimal`).
    - (Necessary loss of precision when converting to **8**-bit grayscale value.)

Continuous Cellular Automata

Would like to produce *images*, not just a *reduction*.

Back to requiring $\boldsymbol{O(SN)}$ memory to store image?

Continuous Cellular Automata

Would like to produce *images*, not just a *reduction*.

Back to requiring $O(SN)$ memory to store image?

```
// Write all rows and columns of the image to the output stream.
void PJGImage.Writer.write();

// Write the given row slice of the image to the output stream.
void PJGImage.Writer.writeRowSlice(Range theRowRange);

// Write the given column slice of the image to the output stream.
void PJGImage.Writer.writeColSlice(Range theColRange);

// Write the given patch of the image to the output stream.
void PJGImage.Writer.writePatch(Range theRowRange, Range theColRange);
```

`CCASeq.java`

`code/CCASeq.java`

Why doesn't **`static byte`**`[][] pixelmatrix` lead to $O(SN)$ memory?

`CCASeq.java`

`code/CCASeq.java`

Why doesn't **`static byte`**`[][] pixelmatrix` lead to $O(SN)$ memory?

Parallelizing `CCA`:

- What portions of the Continuous CA program are not parallelizable?
- What portions of the Continuous CA program are parallelizable?
  - What parallelization patterns?
- Any synchronization issues?
- Any load-balancing issues?
- Any cache interference?
- Any beneficial cache effects?

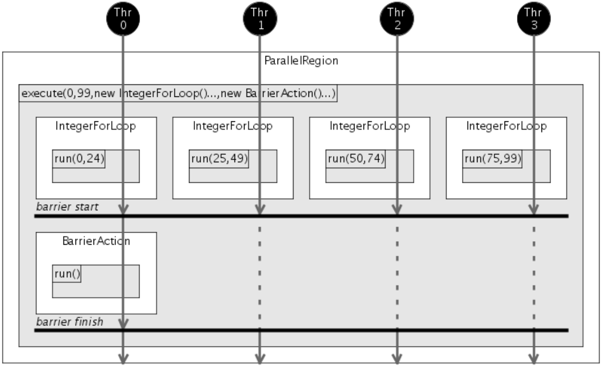
CCASmpAlt.java

code/CCASmpAlt.java

Barrier Actions

```
new ParallelTeam().execute(new ParallelRegion() {
   public void run() {
      ...
      execute(0, 99,
         new IntegerForLoop() {
            public void run(int first, int last) {
               for (int i = first; i <= last; i++) {
                  ... // Loop body
               }
            }
         },
         new BarrierAction() {
           public void run() {
             ... // Code to be executed in a single thread
           }
         });
      ...
   }
});
```

Barrier Actions



Other Barrier Actions

```
// Instead of a barrier action object,
// use a constant BarrierAction:

// Each thread waits at the barrier.
execute(0, 99, new IntegerForLoop { ... },
        BarrierAction.WAIT);

// Each thread does not wait at the barrier.
execute(0, 99, new IntegerForLoop { ... },
        BarrierAction.NO_WAIT);
```

In what situations would `NO_WAIT` be useful?

- Remember: correctness trumps performance

CCASmp.java

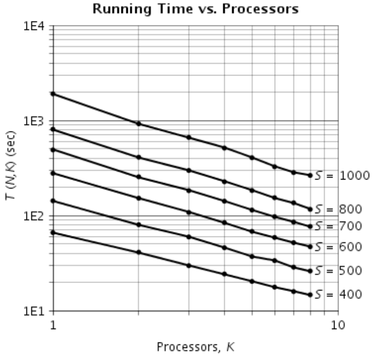
`code/CCASmp.java`

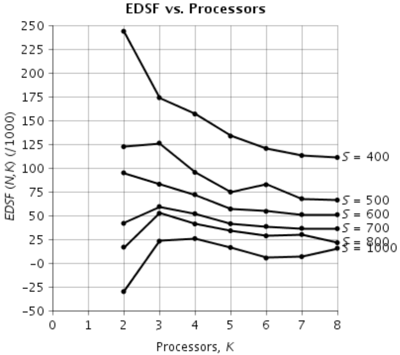
CCASmp.java

code/CCASmp.java

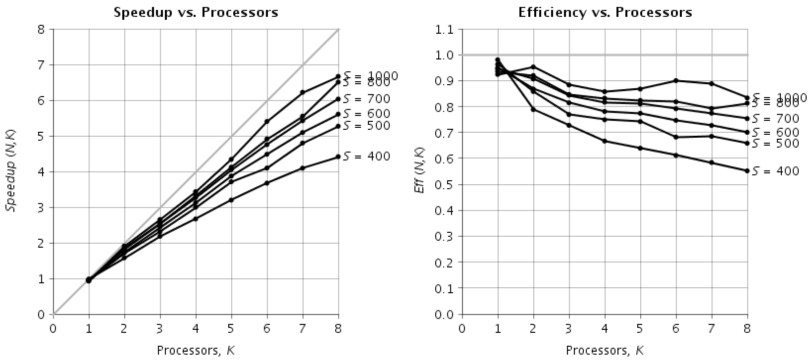
What is the advantage of `CCASmp.java` over `CCASmpAlt.java`?

CCASmp Running Time and EDSF





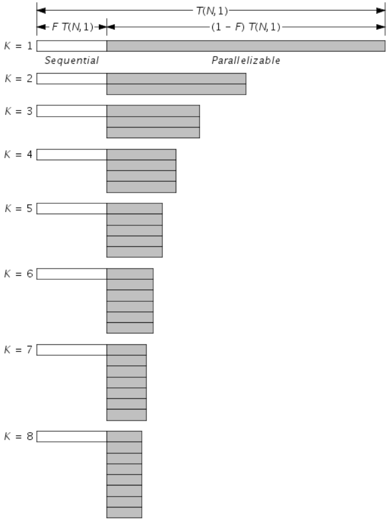
CCASmp Speedup and Efficiency



Classic Amdahl's Law behavior,
but with a rather large sequential fraction.

What portions of the Continuous CA program
are causing the large sequential fraction?

Beating Amdahl's Law



Beating Amdahl's Law with Overlapping

What portions of the Continuous CA program are causing the large sequential fraction?

The barrier action that

- computes grayscale value of cell state
- writes pixel row to image file

Beating Amdahl's Law with Overlapping

What portions of the Continuous CA program are causing the large sequential fraction?

The barrier action that

- computes grayscale value of cell state
- writes pixel row to image file

(Any missed parallelism within these actions?)

Beating Amdahl's Law with Overlapping

What portions of the Continuous CA program
are causing the large sequential fraction?

The barrier action that

- computes grayscale value of cell state
- writes pixel row to image file

(Any missed parallelism within these actions?)

Any missed parallelism between these actions and next-state computation?

Beating Amdahl's Law with Overlapping

What portions of the Continuous CA program are causing the large sequential fraction?

The barrier action that

- computes grayscale value of cell state
- writes pixel row to image file

(Any missed parallelism within these actions?)

Any missed parallelism between these actions and next-state computation?

- Could write the current state to image file while computing the next state.

Beating Amdahl's Law with Overlapping

What portions of the Continuous CA program
are causing the large sequential fraction?

The barrier action that

- computes grayscale value of cell state
- writes pixel row to image file

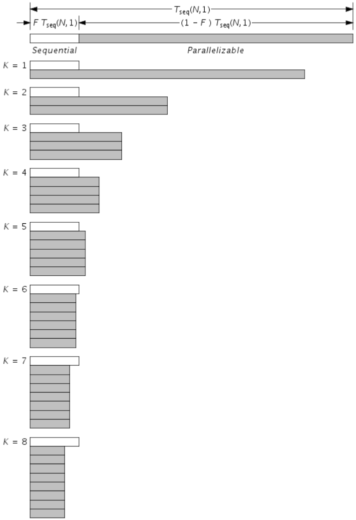
(Any missed parallelism within these actions?)

Any missed parallelism between these actions and next-state computation?

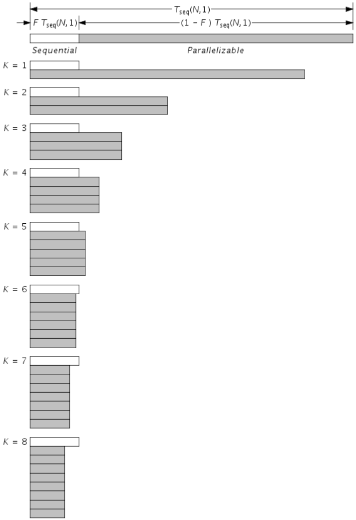
- Could write the current state to image file
  while computing the next state.

Overlapping: run the I/O thread in parallel with the computation threads.

Beating Amdahl's Law with Overlapping



Beating Amdahl's Law with Overlapping



Are we *really* beating Amdahl's Law?

Overlapping: Speedup and Efficiency

Running Time

- $T(N, K) = \max(\ F \cdot T(N, 1)\ ,\ \frac{1}{K} \cdot (1 - F) \cdot T(N, 1)\ )$

Speedup

Overlapping: Speedup and Efficiency

Running Time

- $T(N, K) = \max(\ F \cdot T(N, 1)\ ,\ \frac{1}{K} \cdot (1 - F) \cdot T(N, 1)\ )$

Speedup

- $Speedup(N, K) = \frac{T(N,1)}{T(N,K)} = \min(\ \frac{1}{F}\ ,\ \frac{K}{1-F}\ )$

Efficiency

Overlapping: Speedup and Efficiency

Running Time

- $T(N,K) = \max(\ F \cdot T(N,1)\ ,\ \frac{1}{K} \cdot (1-F) \cdot T(N,1)\ )$

Speedup

- $Speedup(N,K) = \frac{T(N,1)}{T(N,K)} = \min(\ \frac{1}{F}\ ,\ \frac{K}{1-F}\ )$

Efficiency

- $Eff(N,K) = \frac{Speedup(N,K)}{K} = \min(\ \frac{1}{K \cdot F}\ ,\ \frac{1}{1-F}\ )$

Overlapping: Speedup and Efficiency

Running Time

- $T(N,K) = \max(\ F \cdot T(N,1)\ ,\ \frac{1}{K} \cdot (1-F) \cdot T(N,1)\ )$

Speedup

- $\mathit{Speedup}(N,K) = \frac{T(N,1)}{T(N,K)} = \min(\ \frac{1}{F}\ ,\ \frac{K}{1-F}\ )$

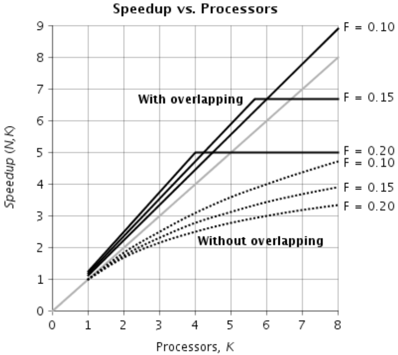
Efficiency

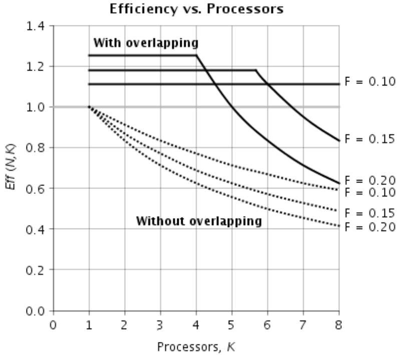
- $\mathit{Eff}(N,K) = \frac{\mathit{Speedup}(N,K)}{K} = \min(\ \frac{1}{K \cdot F}\ ,\ \frac{1}{1-F}\ )$

What happens to ***Speedup*** and ***Eff*** for small K?

What happens to ***Speedup*** and ***Eff*** as $K \to \infty$?

Overlapping: Speedup and Efficiency



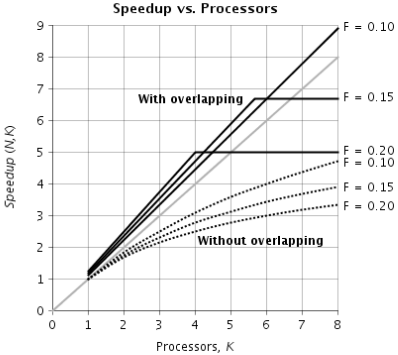


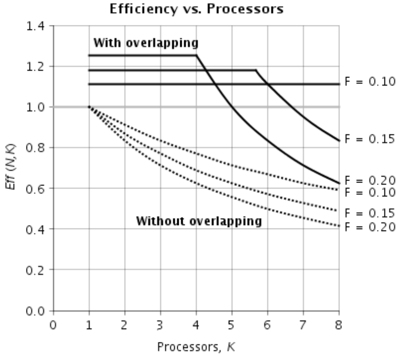
$$\min\left(\frac{1}{F} , \frac{K}{1-F} \right)$$

$$\min\left(\frac{1}{F \cdot K} , \frac{1}{1-F} \right)$$

In the limit, as $K \rightarrow \infty$?

Overlapping: Speedup and Efficiency





$$\min\left(\frac{1}{F} , \frac{K}{1-F} \right) \qquad \min\left(\frac{1}{F \cdot K} , \frac{1}{1-F} \right)$$

In the limit, as $K \rightarrow \infty$?

I claim: this superlinear speedup is "cheating". Why?

Overlapping: Speedup and Efficiency

Running Time

- $T(N,K) = \max(\ F \cdot T(N,1)\ ,\ \frac{1}{K-1} \cdot (1-F) \cdot T(N,1)\ )$

Speedup

- $\mathit{Speedup}(N,K) = \frac{T(N,1)}{T(N,K)} = \min(\ \frac{1}{F}\ ,\ \frac{K-1}{1-F}\ )$

Efficiency

- $\mathit{Eff}(N,K) = \frac{\mathit{Speedup}(N,K)}{K} = \min(\ \frac{1}{K \cdot F}\ ,\ \frac{1}{1-F} - \frac{1}{K \cdot (1-F)}\ )$

What happens to ***Speedup*** and ***Eff*** for small K?

What happens to ***Speedup*** and ***Eff*** as $K \to \infty$?

Overlapping: Speedup and Efficiency

Running Time

- $T(N, K) = \max(\ F \cdot T(N, 1)\ ,\ \frac{1}{K-1} \cdot (1 - F) \cdot T(N, 1)\ )$

Speedup

- $Speedup(N, K) = \frac{T(N,1)}{T(N,K)} = \min(\ \frac{1}{F}\ ,\ \frac{K-1}{1-F}\ )$

Efficiency

- $Eff(N, K) = \frac{Speedup(N,K)}{K} = \min(\ \frac{1}{K \cdot F}\ ,\ \frac{1}{1-F} - \frac{1}{K \cdot (1-F)}\ )$

What happens to ***Speedup*** and ***Eff*** for small K?

What happens to ***Speedup*** and ***Eff*** as $K \to \infty$?

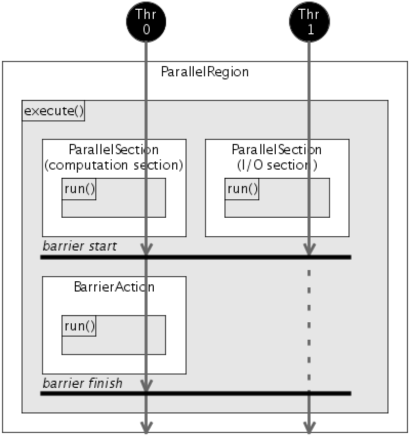
No superlinear speedup.

Parallel Sections

Run a parallel team of threads,
where each thread may execute different code.

```
new ParallelTeam(2).execute(new ParallelRegion() {
  public void run() {
    execute (
      new ParallelSection() {
        public void run() {
          // Code for computation
        }
      },
      new ParallelSection() {
        public void run() {
          // Code for I/O
        }
      },
      new BarrierAction() {
        public void run() {
          // Code for single-threaded barrier action
        }
      });
  }
});
```

Parallel Sections

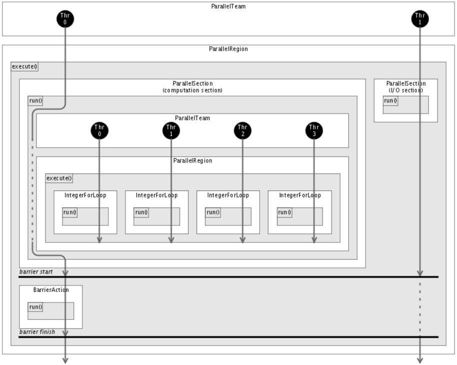


Nested Parallel Regions

In the 1-D CCA program, the computation task is a parallel problem.
The computation section contains another (nested) parallel region.

Nested Parallel Regions

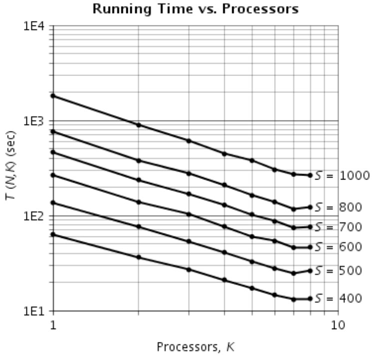
In the 1-D CCA program, the computation task is a parallel problem. The computation section contains another (nested) parallel region.

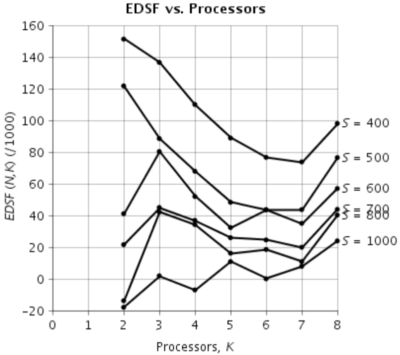


CCASmp2.java

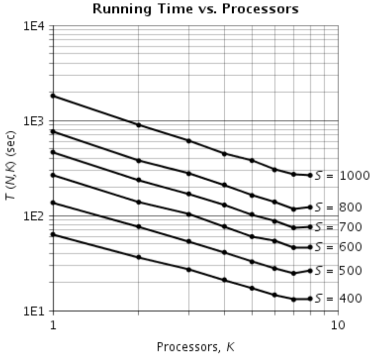
code/CCASmp2.java

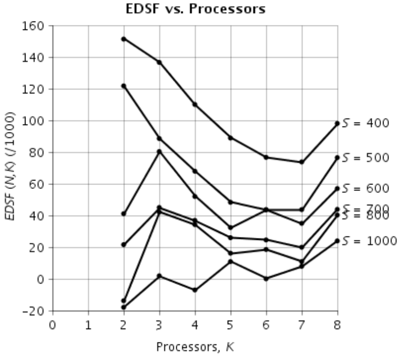
CCASmp2 Running Time and EDSF





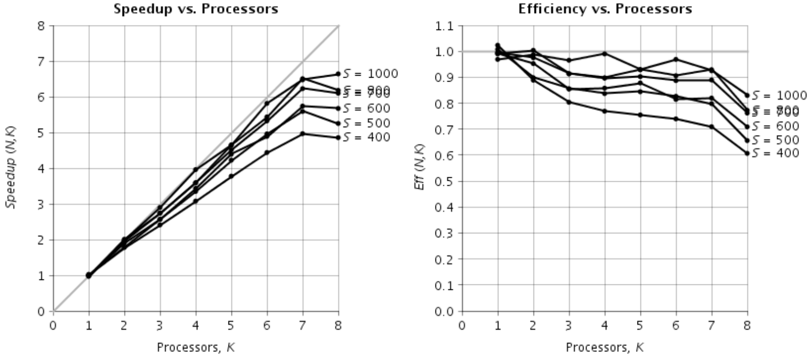
CCASmp2 Running Time and EDSF



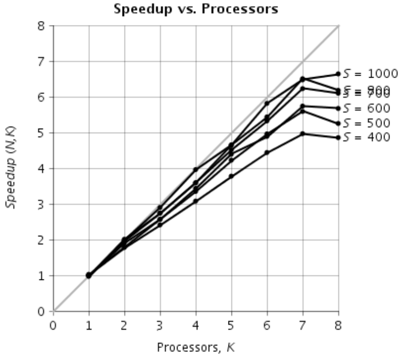


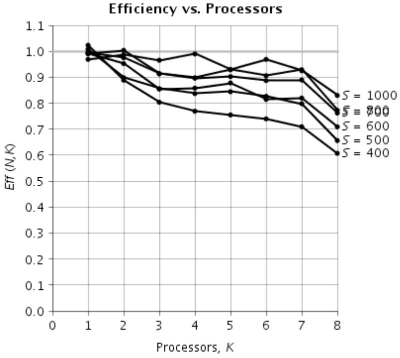
Why the spike at $K = 8$?

CCASmp2 Speedup and Efficiency



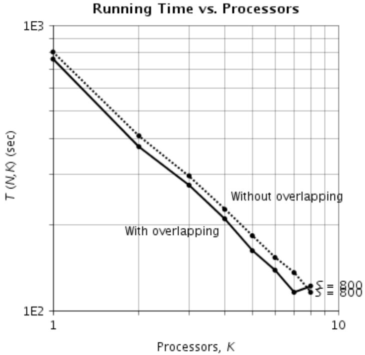
CCASmp2 Speedup and Efficiency

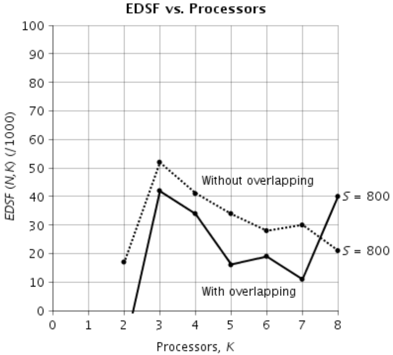




Why the dip at $K = 8$?

CCASmp vs. CCASmp2 Running Time and EDSF





CCASmp vs. CCASmp2 Speedup and Efficiency