Parallel Computing I

Performance Metrics

Performance Metrics

Why parallel computing?

- faster answers
- bigger problems

Performance Metrics

Why parallel computing?

- faster answers
- bigger problems

Begs the questions:

- how much faster?
- how much bigger?

Want to define "faster" and "bigger" in a precise manner.

First, need some definitions:

- Problem size
- Running time
- Speed

Definitions

- Problem size (N): the number of computations the program performs to solve the problem.
  - AES Key Search: $N = 2^n$, where n is number of unknown bits
  - Image Generation: $N = n * m$, where n and m are the dimensions
  - Running time should be proportional to N
- Running Time (T): the amount of time the program takes to compute the answer to the problem.
  - Depends upon HW, algorithm, implementation, etc.
  - $T_{\mathbf{seq}}(N, K)$ and $T_{\mathbf{par}}(N, K)$: emphasizes running time is a function of the implementation ($\mathbf{seq}$ or $\mathbf{par}$), the problem size N and the number of processors K.
- Speed ($S(N, K)$): the rate at which program runs can be done.

$$S(N, K) = \frac{1}{T(N, K)}$$

Speedup

- Speedup $\boldsymbol{Speedup(N,K)}$: the speed of the *parallel* version running on $\boldsymbol{K}$ processors relative to the speed of the *sequential* version running on one processor for a given problem size $\boldsymbol{N}$

$$\boldsymbol{Speedup(N,K) = \frac{S_{\mathrm{par}}(N,K)}{S_{\mathrm{seq}}(N,1)} = \frac{T_{\mathrm{seq}}(N,1)}{T_{\mathrm{par}}(N,K)}}$$

Speedup

- Speedup $\boldsymbol{Speedup(N,K)}$: the speed of the *parallel* version running on $\boldsymbol{K}$ processors relative to the speed of the *sequential* version running on one processor for a given problem size $\boldsymbol{N}$

$$\boldsymbol{Speedup(N,K) = \frac{S_{\mathrm{par}}(N,K)}{S_{\mathrm{seq}}(N,1)} = \frac{T_{\mathrm{seq}}(N,1)}{T_{\mathrm{par}}(N,K)}}$$

  - Why do we not define ***Speedup*** as follows:

$$\boldsymbol{Speedup(N,K) = \frac{S_{\mathrm{par}}(N,K)}{S_{\mathrm{par}}(N,1)} = \frac{T_{\mathrm{par}}(N,1)}{T_{\mathrm{par}}(N,K)}}$$

Speedup

- Speedup $\boldsymbol{Speedup(N, K)}$: the speed of the *parallel* version running on $\boldsymbol{K}$ processors relative to the speed of the *sequential* version running on one processor for a given problem size $\boldsymbol{N}$

$$\boldsymbol{Speedup(N, K) = \frac{S_{\mathrm{par}}(N, K)}{S_{\mathrm{seq}}(N, 1)} = \frac{T_{\mathrm{seq}}(N, 1)}{T_{\mathrm{par}}(N, K)}}$$

Ideally, a parallel program should:

- run twice as fast on two processors (as on one processor)
  - $\boldsymbol{T_{\mathrm{par}}(N, 2) = T_{\mathrm{seq}}(N, 1)/2}$
- run four times as fast on four processors
  - $\boldsymbol{T_{\mathrm{par}}(N, 4) = T_{\mathrm{seq}}(N, 1)/4}$
- ...

Speedup

- Speedup $\boldsymbol{Speedup(N,K)}$: the speed of the *parallel* version running on K processors relative to the speed of the *sequential* version running on one processor for a given problem size N

$$Speedup(N,K) = \frac{S_{\mathrm{par}}(N,K)}{S_{\mathrm{seq}}(N,1)} = \frac{T_{\mathrm{seq}}(N,1)}{T_{\mathrm{par}}(N,K)}$$

Ideally, a parallel program should:

- run twice as fast on two processors (as on one processor)
  - $T_{\mathrm{par}}(N,2) = T_{\mathrm{seq}}(N,1)/2$
- run four times as fast on four processors
  - $T_{\mathrm{par}}(N,4) = T_{\mathrm{seq}}(N,1)/4$
- ...
- $T_{\mathrm{par}}(N,K) = T_{\mathrm{seq}}(N,1)/K$

Speedup

- Speedup $\boldsymbol{Speedup(N, K)}$: the speed of the *parallel* version running on K processors relative to the speed of the *sequential* version running on one processor for a given problem size $\boldsymbol{N}$

$$Speedup(N, K) = \frac{S_{\mathrm{par}}(N, K)}{S_{\mathrm{seq}}(N, 1)} = \frac{T_{\mathrm{seq}}(N, 1)}{T_{\mathrm{par}}(N, K)}$$

Ideally, a parallel program should:

- run twice as fast on two processors (as on one processor)
  - $\boldsymbol{T_{\mathrm{par}}(N, 2) = T_{\mathrm{seq}}(N, 1)/2}$
- run four times as fast on four processors
  - $\boldsymbol{T_{\mathrm{par}}(N, 4) = T_{\mathrm{seq}}(N, 1)/4}$
- ...
- $T_{\mathrm{par}}(N, K) = T_{\mathrm{seq}}(N, 1)/K$ and $Speedup(N, K) = K$

Speedup

- Speedup $\boldsymbol{Speedup(N,K)}$: the speed of the *parallel* version running on $\boldsymbol{K}$ processors relative to the speed of the *sequential* version running on one processor for a given problem size $\boldsymbol{N}$

$$\boldsymbol{Speedup(N,K) = \frac{S_{\mathrm{par}}(N,K)}{S_{\mathrm{seq}}(N,1)} = \frac{T_{\mathrm{seq}}(N,1)}{T_{\mathrm{par}}(N,K)}}$$

Ideally, a parallel program should:

- run twice as fast on two processors (as on one processor)
  - $\boldsymbol{T_{\mathrm{par}}(N,2) = T_{\mathrm{seq}}(N,1)/2}$
- run four times as fast on four processors
  - $\boldsymbol{T_{\mathrm{par}}(N,4) = T_{\mathrm{seq}}(N,1)/4}$
- ...
- $\boldsymbol{T_{\mathrm{par}}(N,K) = T_{\mathrm{seq}}(N,1)/K}$ and $\boldsymbol{Speedup(N,K) = K}$

But, we don't live in an ideal world.

Efficiency

Why is it not the case that $\boldsymbol{Speedup(N, K) = K}$?
How will $\boldsymbol{Speedup(N, K)}$ usually compare to $\boldsymbol{K}$?

Efficiency

Why is it not the case that $\boldsymbol{Speedup}(N, K) = K$?
How will $\boldsymbol{Speedup}(N, K)$ usually compare to K?

- Efficiency ($\boldsymbol{Eff}(N, K)$): a metric that captures how close a program's speedup is to ideal

$$\boldsymbol{Eff}(N, K) = \frac{\boldsymbol{Speedup}(N, K)}{K}$$

  - Usually, $\boldsymbol{Eff}(N, K) < 1$ (sublinear speedup)

Efficiency

Why is it not the case that $\boldsymbol{Speedup(N, K) = K}$?
How will $\boldsymbol{Speedup(N, K)}$ usually compare to $\boldsymbol{K}$?

- Efficiency ($\boldsymbol{Eff(N, K)}$): a metric that captures how close a program's speedup is to ideal

$$\boldsymbol{Eff(N, K) = \frac{Speedup(N, K)}{K}}$$

  - Usually, $\boldsymbol{Eff(N, K) < 1}$ (sublinear speedup)

Helps to know what speedup and efficiency are "supposed" to look like.

- When a program's speedup and efficiency don't follow the right trends, it might be an indication that there is a problem in the program's design or implementation.

Efficiency

Why is it not the case that $\textit{Speedup}(N, K) = K$?
How will $\textit{Speedup}(N, K)$ usually compare to K?

- Efficiency ($\textit{Eff}(N, K)$): a metric that captures how close a program's speedup is to ideal

$$\textit{Eff}(N, K) = \frac{\textit{Speedup}(N, K)}{K}$$

  - Usually, $\textit{Eff}(N, K) < 1$ (sublinear speedup)

Helps to know what speedup and efficiency are "supposed" to look like.

- When a program's speedup and efficiency don't follow the right trends, it might be an indication that there is a problem in the program's design or implementation.

But, first, what are programs "supposed" to look like.

Amdahl's Law

Gene Amdahl's insight:

- a certain portion of any program must be executed sequentially
  - initialization, finalization, synchronization, I/O, etc.
- sequential portion inherently limits the speedup

- Sequential fraction (F): the fraction of a program that must be executed sequentially

Amdahl's Law

Gene Amdahl's insight:

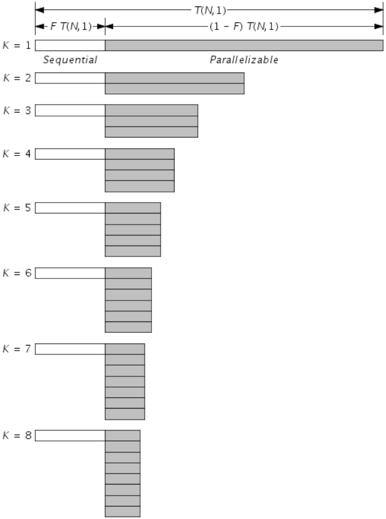
- a certain portion of any program must be executed sequentially
  - initialization, finalization, synchronization, I/O, etc.
- sequential portion inherently limits the speedup

- Sequential fraction (F): the fraction of a program that must be executed sequentially

Amdahl's Law

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

Amdahl's Law



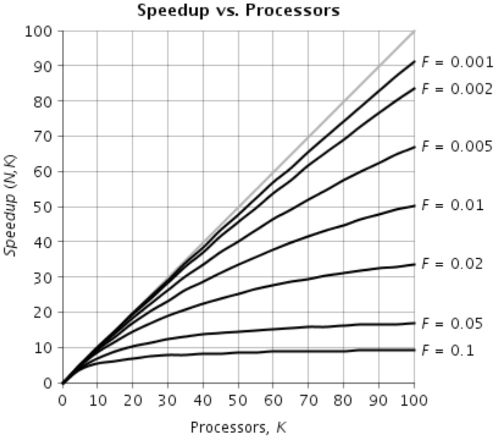
T(N, 1)
F T(N, 1)
(1 − F) T(N, 1)
K = 1
Sequential
Parallelizable
K = 2
K = 3
K = 4
K = 5
K = 6
K = 7
K = 8

Speedup

Speedup as a function of F and K:

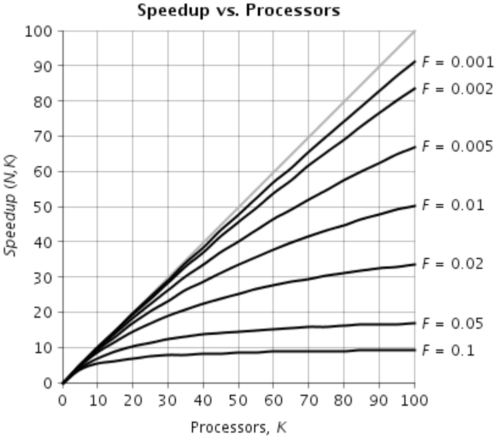
Speedup

Speedup as a function of F and K:



Speedup

Speedup as a function of F and K:



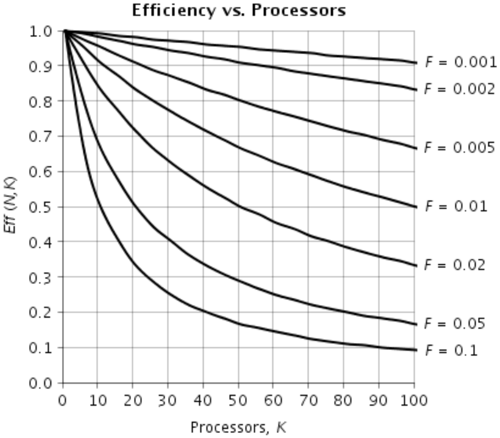
In the limit, as $K \rightarrow \infty$?

Efficiency

Efficiency as a function of F and K:

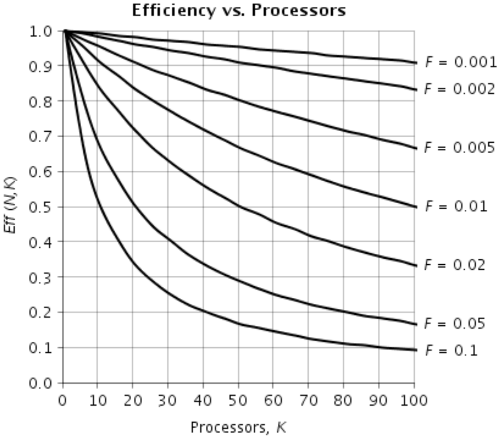
Efficiency

Efficiency as a function of F and K:



Efficiency

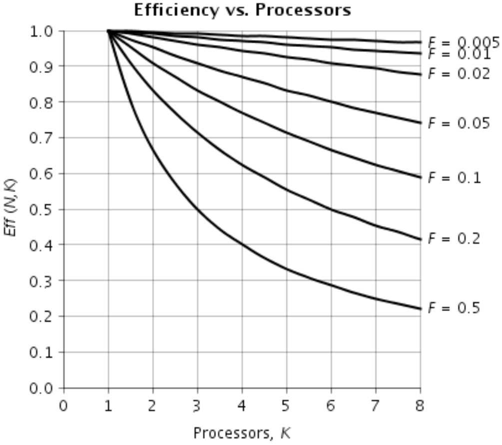
Efficiency as a function of F and K:



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

Consequences of Amdahl's Law

- ***F*** must be very small to acheive good speedup and efficiency as the number of processors increases
- Efficiency vs. processors should resemble:



  - Resemblance can be quantified: what is the ***F***?

Experimentally Determined Sequential Fraction

- Experimentally Determined Sequential Fraction ($\boldsymbol{EDSF(N, K)}$): the sequential fraction $\boldsymbol{F}$ of a program determined using experimental data (running times for a variety of problem sizes and processors)

Experimentally Determined Sequential Fraction

- Experimentally Determined Sequential Fraction ($\boldsymbol{EDSF(N,K)}$): the sequential fraction $\boldsymbol{F}$ of a program determined using experimental data (running times for a variety of problem sizes and processors)

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

  - If the program follows Amdahl's Law, then $\boldsymbol{EDSF}$ (for a fixed problem size) should be constant (as a function of $\boldsymbol{K}$).

How do we obtain experimental data?

Experimental Data

To fully analyze a parallel program, we need to vary a number of things:

- Number of processors
  - Generally easy to set number of processors
  - Very easy to measure
- Problem size
  - Sometimes easy to set problem size (e.g., AES Key Search)
  - Real-world problems not so easy (e.g., weather simulator)
  - Easy to measure

and we need to measure something:

- Running time
  - Hard to measure accurately

Measuring Running Time

- Why does a program's running time vary each run (even with identical inputs)?

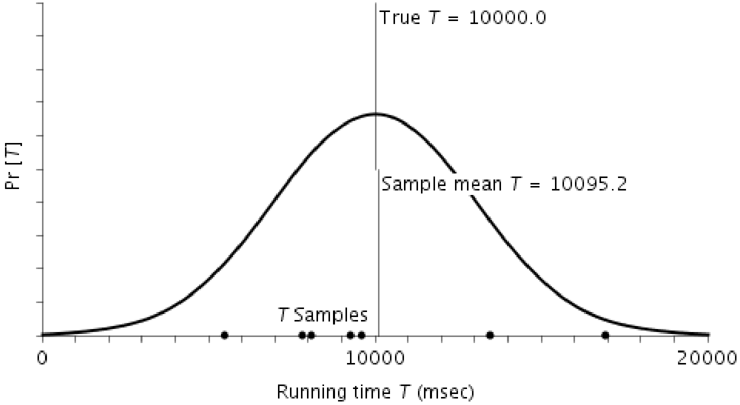
Measuring Running Time

- Why does a program's running time vary each run (even with identical inputs)?
- Need to get a meaningful running time measurement despite all these random(?) fluctuations.

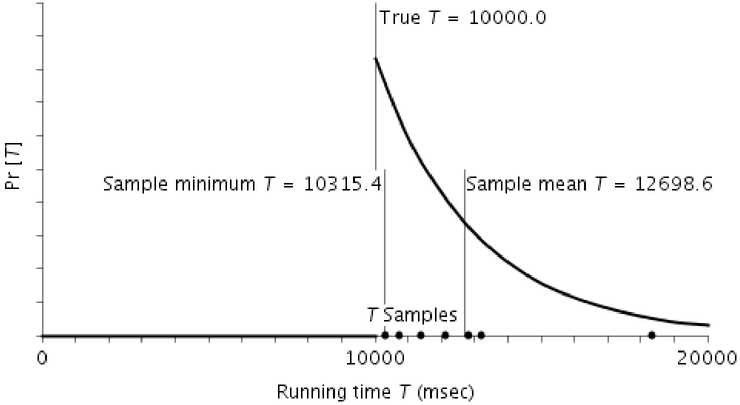
Measuring Running Time

- Why does a program's running time vary each run (even with identical inputs)?
- Need to get a meaningful running time measurement despite all these random(?) fluctuations.
  - average
  - minimum

Measuring Running Time: Avg. vs. Min



Measuring Running Time: Avg. vs. Min



Measuring Running Time

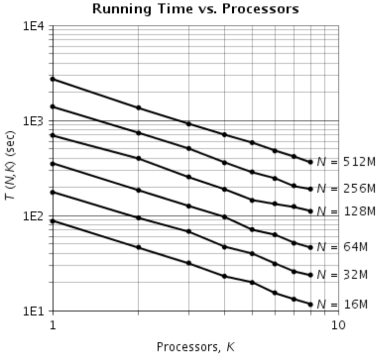
Recommended Approach:

- Use the same machine (or identical hardware) for all program runs.
- Ensure that the program is the only user process running.
- Don't have any server or daemon processes running like Web servers, e-mail servers, file servers, network time daemons, etc.
- Prepare several input data sets covering a range of problem sizes. Choose the smallest problem size so that $T_{\mathbf{seq}}(N, 1)$ is at least $1\mathrm{min}$.
- For each input data set, run the sequential version several times and take the minimum.
- For each input data set and for each number of processors, run the parallel version several times and take the minimum.

Not always easy to acheive in practice. We do our best.

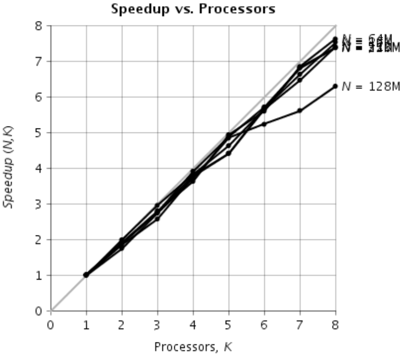
Not clear that results from these experiments translate to "real-world".

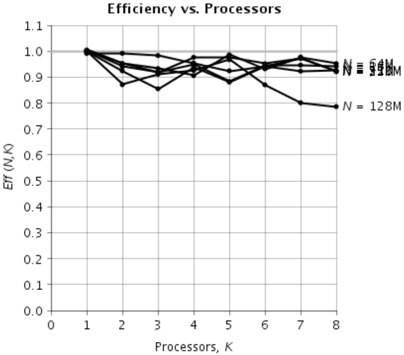
`FindKeySmp` Running Time Results



Note: log-log plot

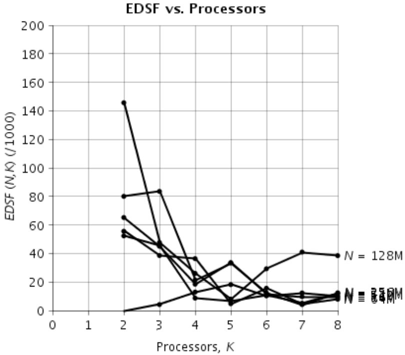
`FindKeySmp` Running Time Results





Eff doesn't resemble desired plot.

`FindKeySmp` Running Time Results



EDSF is nowhere near constant.
`FindKeySmp` isn't just "less-than-ideal",
there may be real design and implementation issues.

Performance Debugging of `FindKeySmp`

Why does the SMP parallel program for AES key search perform poorly?

Performance Debugging of FindKeySmp

Why does the SMP parallel program for AES key search perform poorly?

One answer comes from internal details of the JVM and the CPU.

```
new ParallelTeam().execute(new ParallelRegion() {
   public void run() {
      execute(0,maxcounter,new IntegerForLoop() {
         byte[] trialkey;
         byte[] trialciphertext;
         AES256Cipher cipher;
         public void start() {
            trialkey = new byte[32];
            System.arraycopy(partialkey,0,trialkey,0,32);
            trialciphertext = new byte[16];
            cipher = new AES256Cipher(trialkey);
         }
         ...
```

Performance Debugging of `FindKeySmp`

Why does the SMP parallel program for AES key search perform poorly?

One answer comes from internal details of the JVM and the CPU.

```
new ParallelTeam().execute(new ParallelRegion() {
   public void run() {
      execute(0,maxcounter,new IntegerForLoop() {
         byte[] trialkey;
         byte[] trialciphertext;
         AES256Cipher cipher;
         public void start() {
            trialkey = new byte[32];
            System.arraycopy(partialkey,0,trialkey,0,32);
            trialciphertext = new byte[16];
            cipher = new AES256Cipher(trialkey);
         }
         ...
```

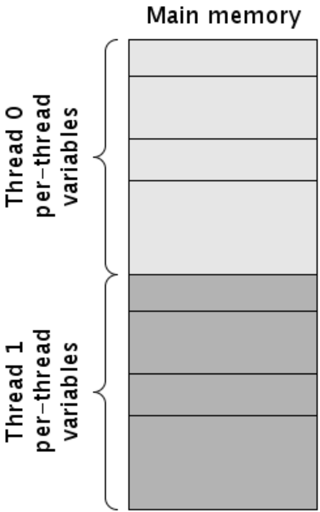
Where/what are the memory allocations?

Memory Allocations

- **byte**`[] trialkey` — 4/8 byte reference, per thread
- **byte**`[] trialciphertext` — 4/8 byte reference, per thread
- `AES256Cipher cipher` — 4/8 byte reference, per thread
- **new byte**`[32]` — 32+ bytes, per thread
- **new byte**`[16]` — 16+ bytes, per thread
- **new** `AES256Cipher(trialkey)` — ??? bytes, per thread

No requirements on JVM regarding where allocations are located in memory,
but objects allocated close in time are likely to be close in space.

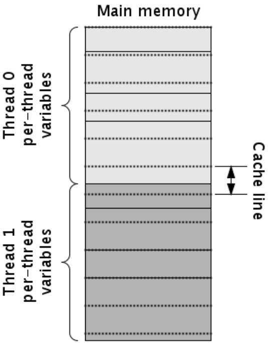
Memory Allocations



Cache Interference

Recall that CPU cache will cache 64 or 128-bytes of contiguous memory.

Problem: var/object boundaries and cache line boundaries may not be aligned.



Cache Interference

What happens if the per-thread variables of two different threads happen to fall in the same cache line?

- when one thread reads?
- when one thread writes?

False sharing of cache lines.

- Threads/CPUs share the cache line, but no data within the cache line.

Eliminating Cache Interference

How can we eliminate cache interference?
Must ensure that different threads' per-thread variables never reside in the same cache line.

Eliminating Cache Interference

How can we eliminate cache interference?
Must ensure that different threads' per-thread variables never reside in the same cache line.

JVM and `javac` could help (but they don't).

Eliminating Cache Interference

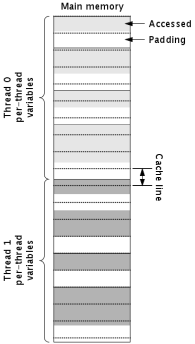
How can we eliminate cache interference?
Must ensure that different threads' per-thread variables never reside in the same cache line.

JVM and `javac` could help (but they don't).

Add padding to per-thread objects:

- 64- or 128-bytes of "dummy" data that is never accessed by thread

Eliminating Cache Interference

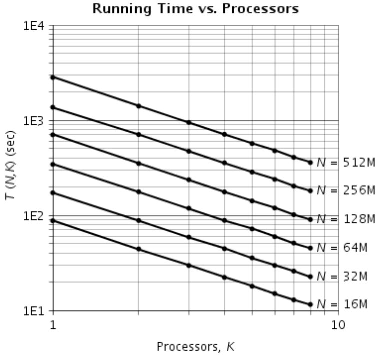


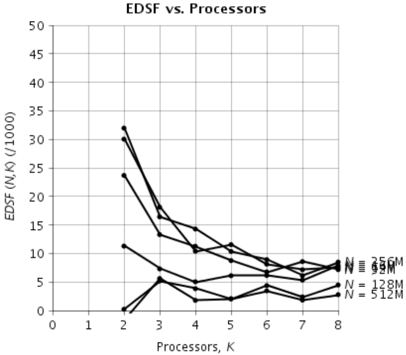
Eliminating Cache Interference

```
new ParallelTeam().execute(new ParallelRegion() {
   public void run() {
      execute(0,maxcounter,new IntegerForLoop() {
         byte[] trialkey;
         byte[] trialciphertext;
         AES256Cipher cipher;
         long p0, p1, p2, p3, p4, p5, p6, p7; // padding
         long p8, p9, pa, pb, pc, pd, pe, pf; // padding
         public void start() {
            trialkey = new byte[32+128]; // + padding
            System.arraycopy(partialkey,0,trialkey,0,32);
            trialciphertext = new byte[16+128]; // + padding
            cipher = new AES256Cipher(trialkey);
         }
         ...
```

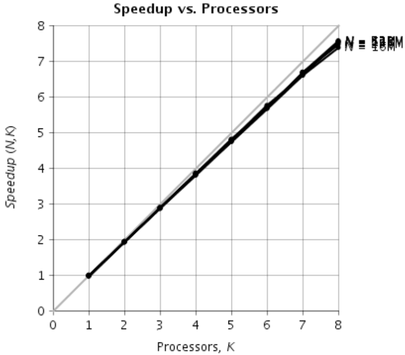
Now "waste" 384 bytes per thread.
But, our parallel performance improves.

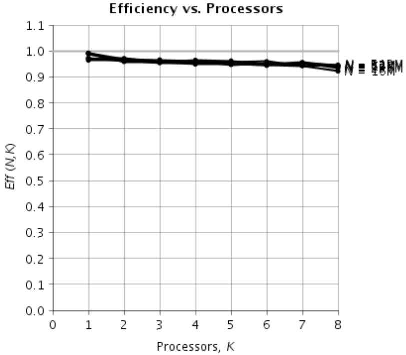
FindKeySmp3 Running Time Results





FindKeySmp3 Running Time Results





Performance Debugging of `FindKeySmp`

Why are there still ***EDSF*** anomalies?

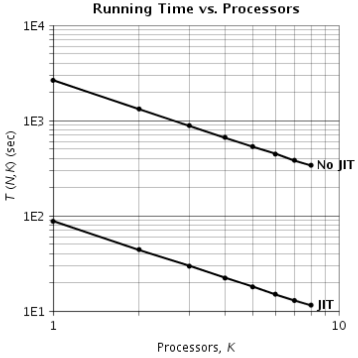
Performance Debugging of `FindKeySmp`

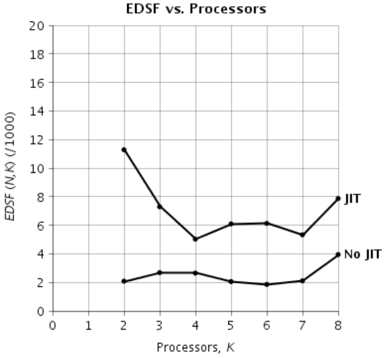
Why are there still ***EDSF*** anomalies?

Another answer comes from internal details of the JVM.

- Just-in-time (JIT) compilation

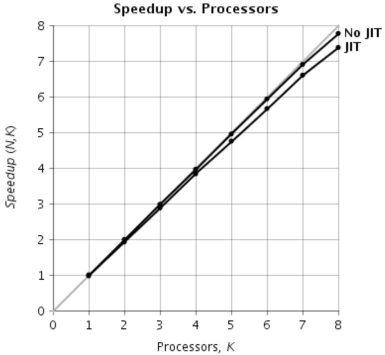
`FindKeySmp3` Running Time Results ($N = 16\mathrm{M}$)

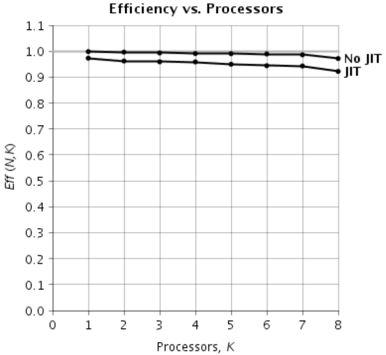




Last-core slowdown?

FindKeySmp3 Running Time Results ($N = 16\mathrm{M}$)





Last-core slowdown?

Performance Metrics

Why parallel computing?

- faster answers
- bigger problems

Begs the questions:

- how much faster?
- how much bigger?

Want to define "faster" and "bigger" in a precise manner.

So far, have focused on "faster".
Now, let's look at "bigger".

Sizeup

Thus far, we have set the problem size and measured the running time. An alternative is to set the running time and measure the problem size.

- Problem size ($N(T, K)$): the problem size for which the running time of the program on K processors will be exactly T.
  - Usually an "impossible" problem size, but a useful guide.

Sizeup and Sizeup Efficiency

- Sizeup $\mathit{Sizeup}(T, K)$: the size of the *parallel* version running on K processors relative to the size of the *sequential* version running on one processor for a given running time T.

$$\mathit{Sizeup}(T, K) = \frac{N_{\mathrm{par}}(T, K)}{N_{\mathrm{seq}}(T, 1)}$$

Ideally, a parallel program should:

- solve twice as big a problem on two processors (as on one processor)
  - $N_{\mathrm{par}}(T, 2) = N_{\mathrm{seq}}(T, 1)/2$
- solve four times as big a problem on four processors
  - $N_{\mathrm{par}}(T, 4) = N_{\mathrm{seq}}(T, 1)/4$
- ...
- $N_{\mathrm{par}}(T, K) = N_{\mathrm{seq}}(T, 1)/K$ and $\mathit{Sizeup}(T, K) = K$

Sizeup and Sizeup Efficiency

- Sizeup $\textit{Sizeup}(T, K)$: the size of the *parallel* version running on K processors relative to the size of the *sequential* version running on one processor for a given running time T.

$$\textit{Sizeup}(T, K) = \frac{N_{\mathrm{par}}(T, K)}{N_{\mathrm{seq}}(T, 1)}$$

- Sizeup Efficiency ($\textit{SizeupEff}(T, K)$): a metric that captures how close a program's sizeup is to ideal

$$\textit{SizeupEff}(T, K) = \frac{\textit{Sizeup}(T, K)}{K}$$

  - Usually, $\textit{SizeupEff}(T, K) < 1$ (sublinear sizeup)

Sizeup and Sizeup Efficiency

- Sizeup $\textit{Sizeup}(T, K)$: the size of the *parallel* version running on K processors relative to the size of the *sequential* version running on one processor for a given running time T.

$$\textit{Sizeup}(T, K) = \frac{N_{\mathrm{par}}(T, K)}{N_{\mathrm{seq}}(T, 1)}$$

- Sizeup Efficiency ($\textit{SizeupEff}(T, K)$): a metric that captures how close a program's sizeup is to ideal

$$\textit{SizeupEff}(T, K) = \frac{\textit{Sizeup}(T, K)}{K}$$

  - Usually, $\textit{SizeupEff}(T, K) < 1$ (sublinear sizeup)

What are sizeup and sizeup efficiency "supposed" to look like?

Gustafson's Law

John Gustafson's observations:

- One does not take a fixed-size problem and run it on various numbers of processors except when doing academic research; in practice, *the problem size scales with the number of processors*.
- As a first approximation, we have found that it is the *parallel* or *vector* part of a program that scales with the problem size.

Gustafson's Law

John Gustafson's observations:

- One does not take a fixed-size problem and run it on various numbers of processors except when doing academic research; in practice, *the problem size scales with the number of processors*.
- As a first approximation, we have found that it is the *parallel* or *vector* part of a program that scales with the problem size.

In other words, the running time for the sequential portion of the program is the same no matter what the problem size.

- Probably not always true (e.g., loading the problem from disk), but perhaps "true enough".

Gustafson's Law

John Gustafson's observations:

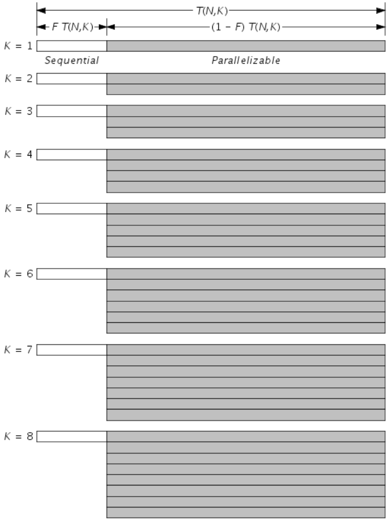
- One does not take a fixed-size problem and run it on various numbers of processors except when doing academic research; in practice, *the problem size scales with the number of processors*.
- As a first approximation, we have found that it is the *parallel* or *vector* part of a program that scales with the problem size.

In other words, the running time for the sequential portion of the program is the same no matter what the problem size.

- Probably not always true (e.g., loading the problem from disk), but perhaps "true enough".

Gustafson recommended that when running a parallel program on a computer with more processors, one should make the problem size larger to keep the running time the same.

Gustafson's Law



$$T(N,1) = F \cdot T(N,K) + K \cdot (1 - F) \cdot T(N,K)$$

Revisiting Speedup and Efficiency w/ Gustafson's Law

Revisiting Speedup and Efficiency w/ Gustafson's Law

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

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

Revisiting Speedup and Efficiency w/ Gustafson's Law

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

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

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

Revisiting Speedup and Efficiency w/ Gustafson's Law

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

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

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

Is there a contradiction between Amdahl's Law and Gustafson's Law?

- As $K \to \infty$, speedup and efficiency behave differently?

Measuring Problem Size

Can't directly measure $\boldsymbol{N(T, K)}$, but we can measure $\boldsymbol{T(N, K)}$.

Measuring Problem Size

Can't directly measure $N(T, K)$, but we can measure $T(N, K)$.

Gustafson assumed sequential portion of program is independent of problem size:

$$T(N, K) = a + \frac{1}{K}(d \cdot N)$$

Measuring Problem Size

Can't directly measure $N(T, K)$, but we can measure $T(N, K)$.

Gustafson assumed sequential portion of program is independent of problem size:

$$T(N, K) = a + \frac{1}{K}(d \cdot N)$$

Consequently:

- $N(T, K) = K \cdot (T - a)/d$
- $Sizeup(T, K) = N(T, K)/N(T, 1) = K$
- $SizeupEff(T, K) = Sizeup(T, K)/K = 1$

Measuring Problem Size

Can't directly measure $N(T, K)$, but we can measure $T(N, K)$.

Better assumption is that sequential portion is proportional to problem size (but at a different rate than parallel portion):

$$T(N, K) = (a + b \cdot N) + \frac{1}{K}(c + d \cdot N)$$

Measuring Problem Size

Can't directly measure $\boldsymbol{N(T, K)}$, but we can measure $\boldsymbol{T(N, K)}$.

Better assumption is that sequential portion is proportional to problem size (but at a different rate than parallel portion):

$$T(N, K) = (a + b \cdot N) + \frac{1}{K}(c + d \cdot N)$$

Consequently:

- $N(T, K) = (K \cdot T - K \cdot a - c)/(K \cdot b + d)$
- $Sizeup(T, K) = (K \cdot T - K \cdot a - c)(b + d)/(T - a - c)(K \cdot b + d)$

Measuring Problem Size

Can't directly measure $N(T, K)$, but we can measure $T(N, K)$.

Better assumption is that sequential portion is proportional to problem size (but at a different rate than parallel portion):

$$T(N, K) = (a + b \cdot N) + \frac{1}{K}(c + d \cdot N)$$

Consequently:

- $N(T, K) = (K \cdot T - K \cdot a - c)/(K \cdot b + d)$
- $Sizeup(T, K) = (K \cdot T - K \cdot a - c)(b + d)/(T - a - c)(K \cdot b + d)$

For large problem sizes N,
the a and c constants won't contribute much to running time T,
so simplify things with $a = 0$ and $c = 0$.

Measuring Problem Size

Can't directly measure $N(T, K)$, but we can measure $T(N, K)$.

Better assumption is that sequential portion is proportional to problem size (but at a different rate than parallel portion):

$$T(N, K) = b \cdot N + \frac{1}{K}(d \cdot N)$$

Measuring Problem Size

Can't directly measure $\boldsymbol{N(T, K)}$, but we can measure $\boldsymbol{T(N, K)}$.

Better assumption is that sequential portion is proportional to problem size (but at a different rate than parallel portion):

$$\boldsymbol{T(N, K) = b \cdot N + \frac{1}{K}(d \cdot N)}$$

Consequently:

- $\boldsymbol{N(T, K) = T/(K \cdot b + d)}$
- $\boldsymbol{Sizeup(T, K) = (K \cdot T)(b + d)/(T)(K \cdot b + d)}$

Measuring Problem Size

Can't directly measure $N(T, K)$, but we can measure $T(N, K)$.

Better assumption is that sequential portion is proportional to problem size (but at a different rate than parallel portion):

$$T(N, K) = b \cdot N + \frac{1}{K}(d \cdot N)$$

Consequently:

- $N(T, K) = T/(K \cdot b + d)$
- $Sizeup(T, K) = (K \cdot T)(b + d)/(T)(K \cdot b + d)$

Let $G = b/d$ (ratio of growth rates).

- $Sizeup(T, K) = (K \cdot G + K)/(K \cdot G + 1)$
- $SizeupEff(T, K) = Sizeup(T, K)/K = (G + 1)/(K \cdot G + 1)$

Measuring Problem Size

Can't directly measure $N(T, K)$, but we can measure $T(N, K)$.

Better assumption is that sequential portion is proportional to problem size (but at a different rate than parallel portion):

$$T(N, K) = b \cdot N + \frac{1}{K}(d \cdot N)$$

Consequently:

- $N(T, K) = T/(K \cdot b + d)$
- $Sizeup(T, K) = (K \cdot T)(b + d)/(T)(K \cdot b + d)$

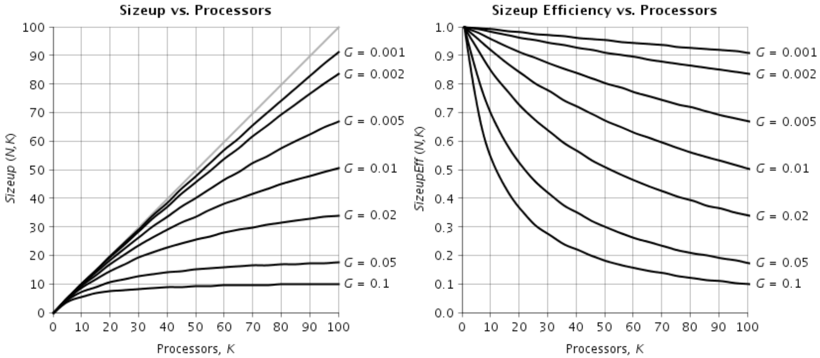
Let $G = b/d$ (ratio of growth rates).

- $Sizeup(T, K) = (K \cdot G + K)/(K \cdot G + 1)$
- $SizeupEff(T, K) = Sizeup(T, K)/K = (G + 1)/(K \cdot G + 1)$

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

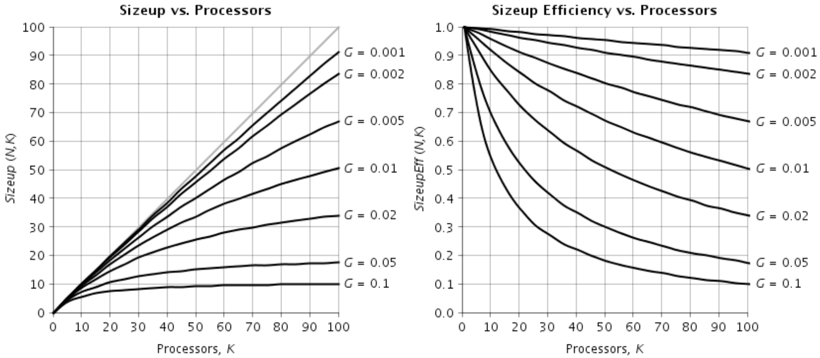
Sizeup and Sizeup Efficiency

Sizeup and Sizeup Efficiency as a function of G and K:



Sizeup and Sizeup Efficiency

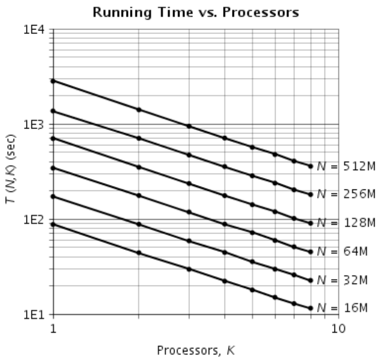
Sizeup and Sizeup Efficiency as a function of G and K:



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

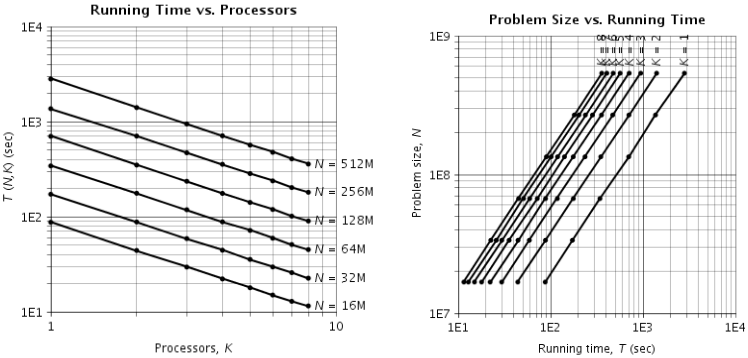
Measuring Problem Size

Can't directly measure $N(T, K)$, but we can measure $T(N, K)$.



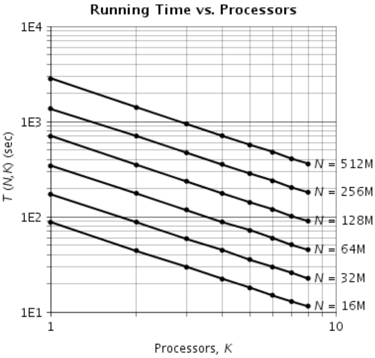
Measuring Problem Size

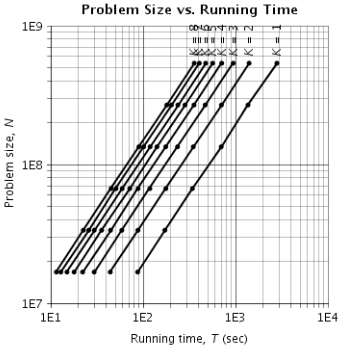
Can't directly measure $\boldsymbol{N(T,K)}$, but we can measure $\boldsymbol{T(N,K)}$.



Measuring Problem Size

Can't directly measure $\boldsymbol{N(T, K)}$, but we can measure $\boldsymbol{T(N, K)}$.

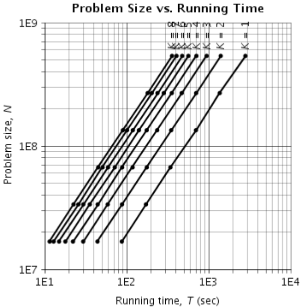




Can we do better than trying to read values off of the plot?

Measuring Problem Size via Linear Interpolation

Can't directly measure $N(T, K)$, but we can measure $T(N, K)$.

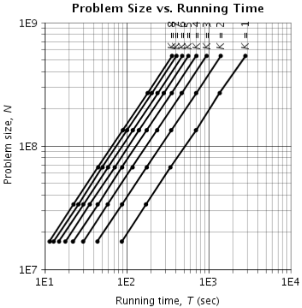


$$N = \frac{T - T_1}{T_2 - T_1}(N_2 - N_1) + N_1$$

where $T_1 \leq T \leq T_2$

Measuring Problem Size via Linear Interpolation

Can't directly measure $N(T, K)$, but we can measure $T(N, K)$.



$$N = \frac{T - T_1}{T_2 - T_1}(N_2 - N_1) + N_1$$

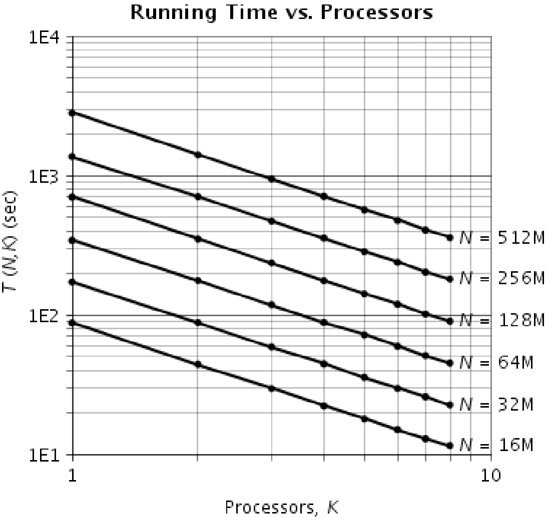
where $T_1 \leq T \leq T_2$

Limitation: Only "accurate" for times with data for all K

- e.g., $\mathbf{90}\mathrm{sec}$ to $\mathbf{350}\mathrm{sec}$
- outside this range, *extrapolating* from experimental data

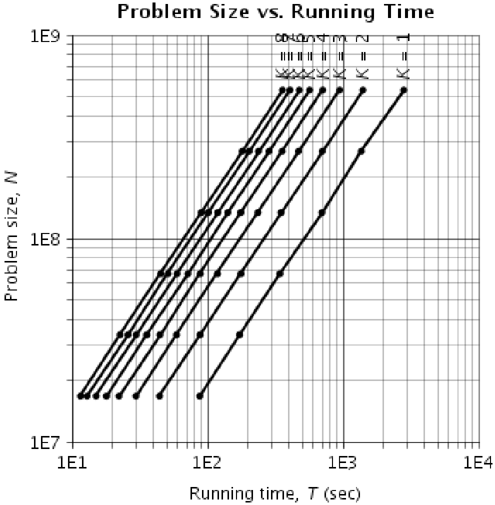
FindKeySmp3 Problem Size Results

Can't directly measure $N(T, K)$, but we can measure $T(N, K)$.



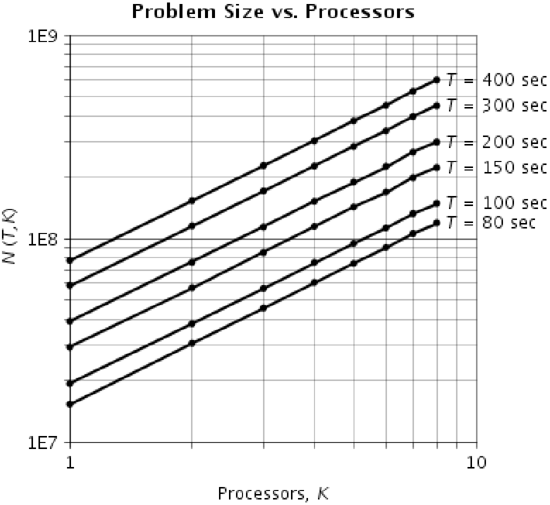
FindKeySmp3 Problem Size Results

Can't directly measure $\boldsymbol{N(T, K)}$, but we can measure $\boldsymbol{T(N, K)}$.

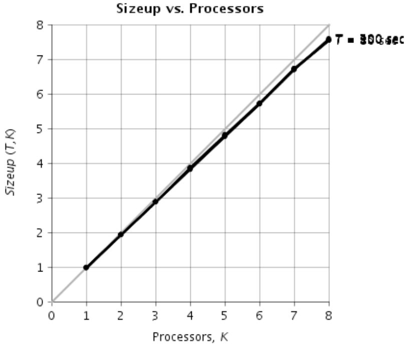


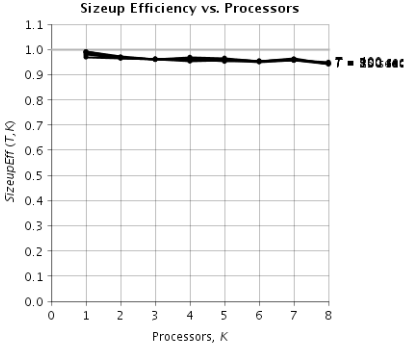
FindKeySmp3 Problem Size Results

Can't directly measure $N(T, K)$, but we can measure $T(N, K)$.



FindKeySmp3 Problem Size Results





Speedup vs. Sizeup

As the number of processors increases:

- faster answers — reduce the running time while keeping the same problem size (speedup)
- bigger problems — increase the problem size while keeping the same running time (sizeup)

Gustafson's observation is that real-world world practioners go for sizeup.

Speedup vs. Sizeup

As the number of processors increases:

- faster answers — reduce the running time while keeping the same problem size (speedup)
- bigger problems — increase the problem size while keeping the same running time (sizeup)

Gustafson's observation is that real-world world practioners go for sizeup.

But, both speedup and sizeup have their limits:

- As $K \to \infty$, $\boldsymbol{Speedup}(N, K) = 1/F$.
- As $K \to \infty$, $\boldsymbol{Sizeup}(T, K) = 1 + 1/G$.

Speedup vs. Sizeup

As the number of processors increases:

- faster answers — reduce the running time while keeping the same problem size (speedup)
- bigger problems — increase the problem size while keeping the same running time (sizeup)

Gustafson's observation is that real-world world practioners go for sizeup.

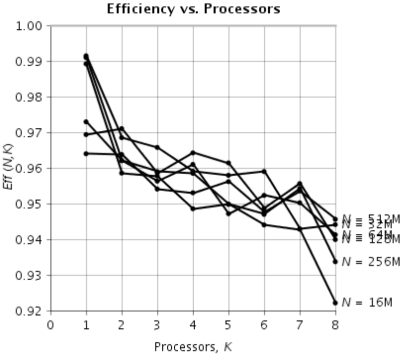
But, both speedup and sizeup have their limits:

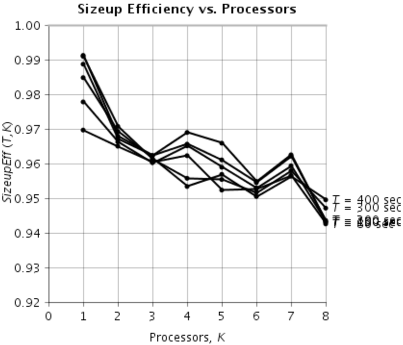
- As $K \to \infty$, $\boldsymbol{Speedup(N, K) = 1/F}$.
- As $K \to \infty$, $\boldsymbol{Sizeup(T, K) = 1 + 1/G}$.

Evidence is that speedup approaches limit faster
than sizeup approaches limit (as number of processors increases).

So, more processors go further with sizeup.

`FindKeySmp3` Efficiency and Sizeup Efficiency





Speedup vs. Sizeup

Speedup is important during the parallel program's *development* stage:

- Speedup is more sensitive to number of processors and sequential fraction.
- Focus on speedup to magnify design or implementation flaws in parallel program's performance.

Sizeup is important during the parallel program's *operational* stage:

- In the field, can increase problem size as processors increase.
- Focus on sizeup now that flaws in parallel program's performance fixed.