SUMMIT

JBoss
WORLD

PRESENTED BY RED HAT

LEARN. NETWORK.
EXPERIENCE OPEN SOURCE.

www.theredhatsummit.com

Simplifying Parallel Programming

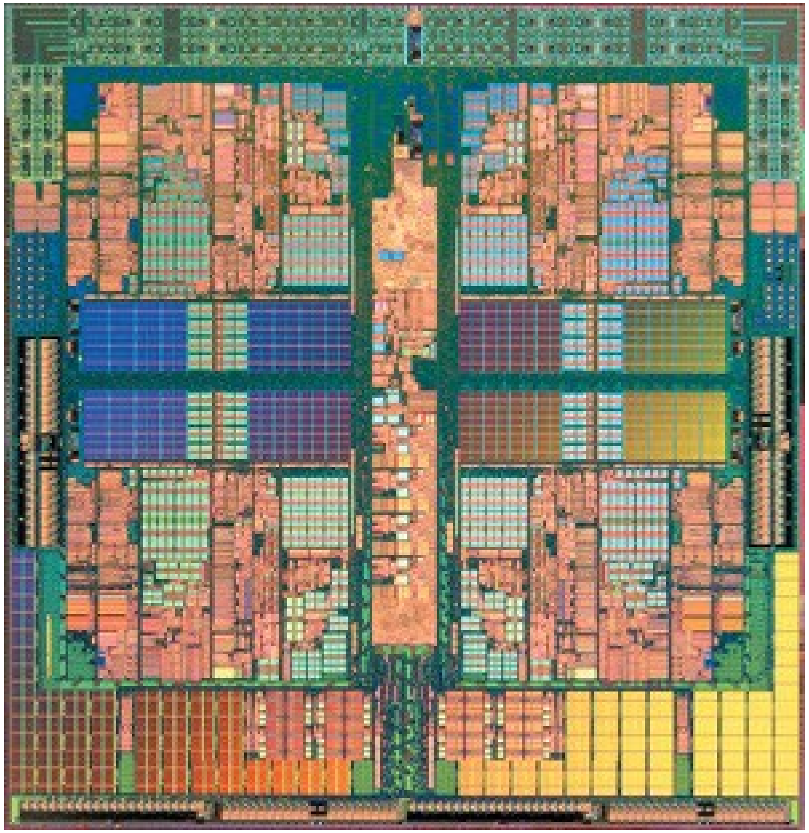
Ulrich Drepper
Consulting Engineer, Red Hat

2010-6-25

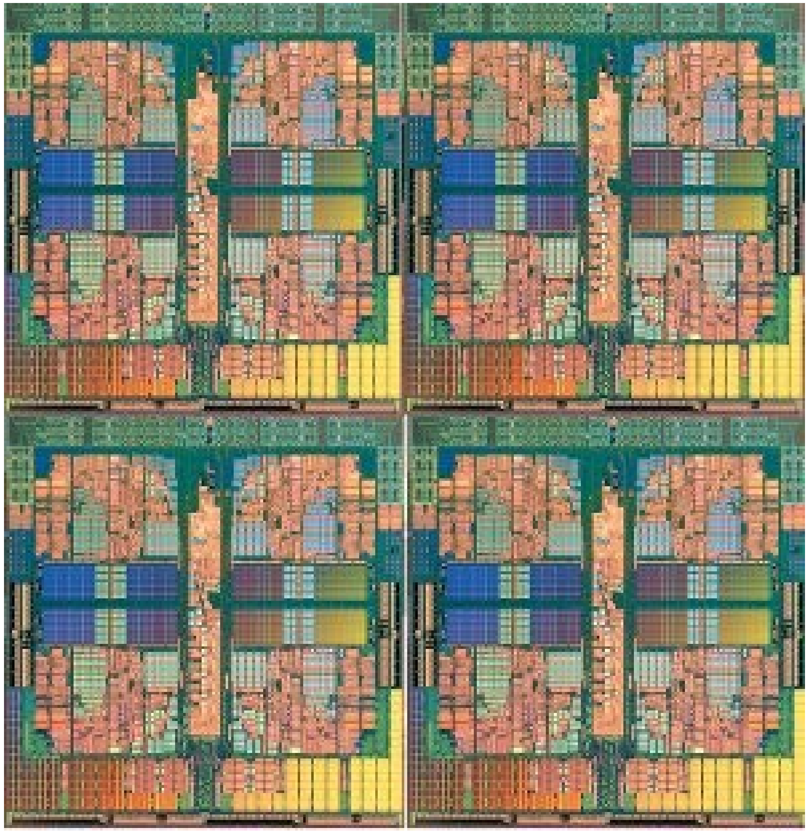
SUMMIT JBoss WORLD

PRESENTED BY RED HAT

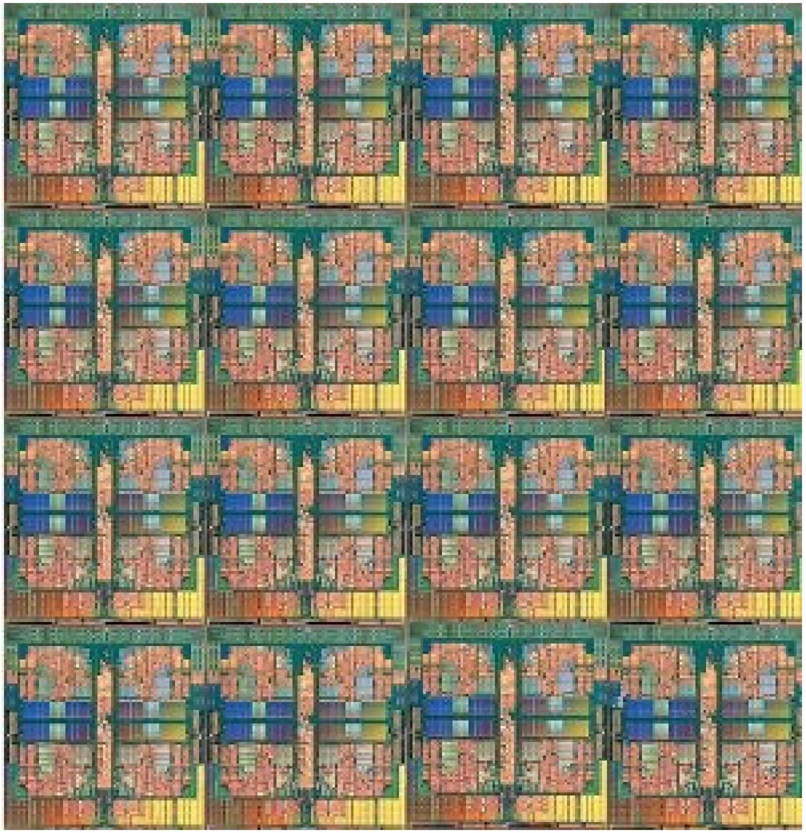
The Problem



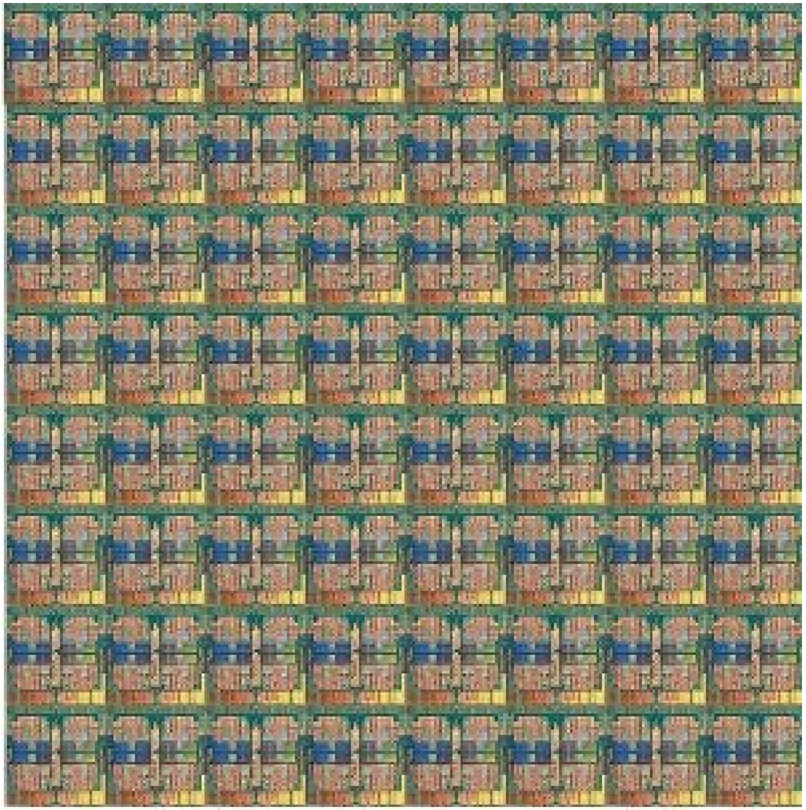
The Problem



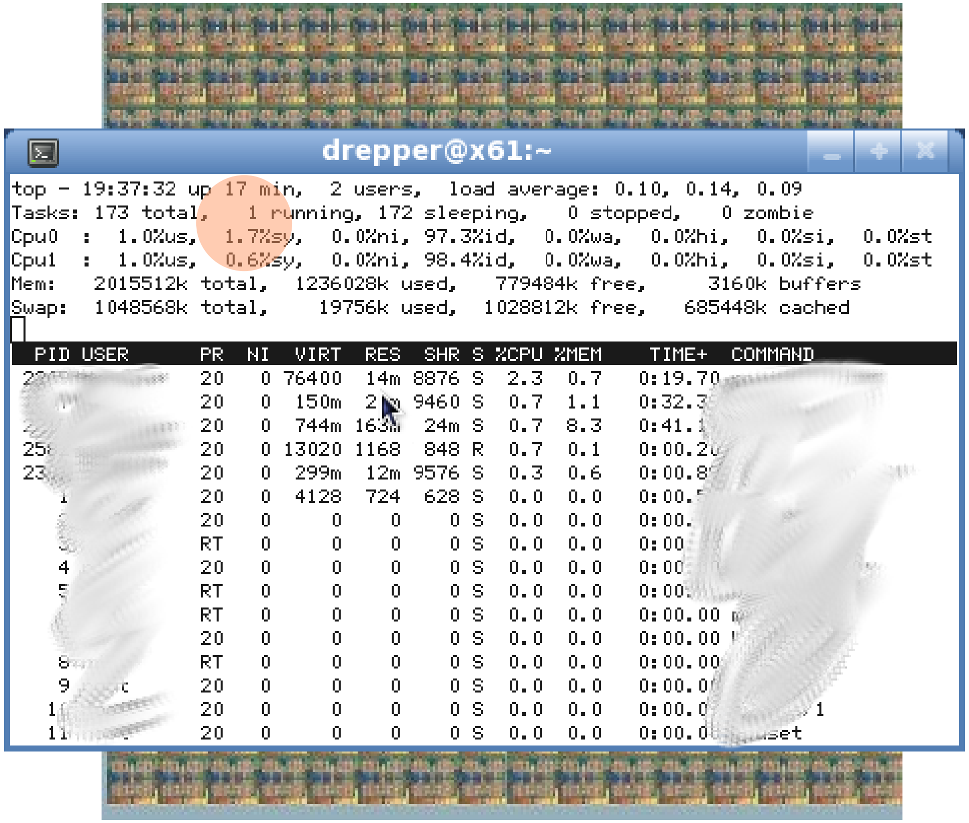
The Problem



The Problem



The Problem



```
top - 19:37:32 up 17 min,  2 users,  load average: 0.10, 0.14, 0.09
Tasks: 173 total,   1 running, 172 sleeping,   0 stopped,   0 zombie
Cpu0  :  1.0%us,  1.7%sy,  0.0%ni, 97.3%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :  1.0%us,  0.6%sy,  0.0%ni, 98.4%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   2015512k total,  1236028k used,   779484k free,     3160k buffers
Swap:  1048568k total,    19756k used,  1028812k free,   685448k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 22             20   0 76400  14m 8876 S  2.3  0.7   0:19.70
                20   0  150m  2 m 9460 S  0.7  1.1   0:32.3
                20   0  744m 163m  24m S  0.7  8.3   0:41.1
 25             20   0 13020 1168  848 R  0.7  0.1   0:00.2
 23             20   0  299m  12m 9576 S  0.3  0.6   0:00.8
    1           20   0  4128  724  628 S  0.0  0.0   0:00.5
                20   0     0    0    0 S  0.0  0.0   0:00.
                RT   0     0    0    0 S  0.0  0.0   0:00
    4           20   0     0    0    0 S  0.0  0.0   0:00
    5           RT   0     0    0    0 S  0.0  0.0   0:00.
                RT   0     0    0    0 S  0.0  0.0   0:00.00
                20   0     0    0    0 S  0.0  0.0   0:00.00
    8           RT   0     0    0    0 S  0.0  0.0   0:00.00
    9           20   0     0    0    0 S  0.0  0.0   0:00.0
   1            20   0     0    0    0 S  0.0  0.0   0:00.0        1
   11           20   0     0    0    0 S  0.0  0.0   0:00.0      set
```

The Reason

$$E = C \times V^2 \times f$$

More Correctly

$$E = C \times V(f)^2 \times f$$

Use of Transistors

- Increasing frequency is out
- Two uses
  - More complex architecture
    - Handle existing instructions faster
    - More specialized instructions
  - Horizontal growth
    - More execution cores; or
    - Only more execution contexts

Requires Parallelism!

Cost of Too Little Parallelism

- Idealized Amdahl's Law

$$S = \frac{1}{(1-\boldsymbol{P}) + \dfrac{\boldsymbol{P}}{N}}$$

- Problems
  - ***P*** too small
  - *N* is steadily growing
- Formula is unrealistic though…

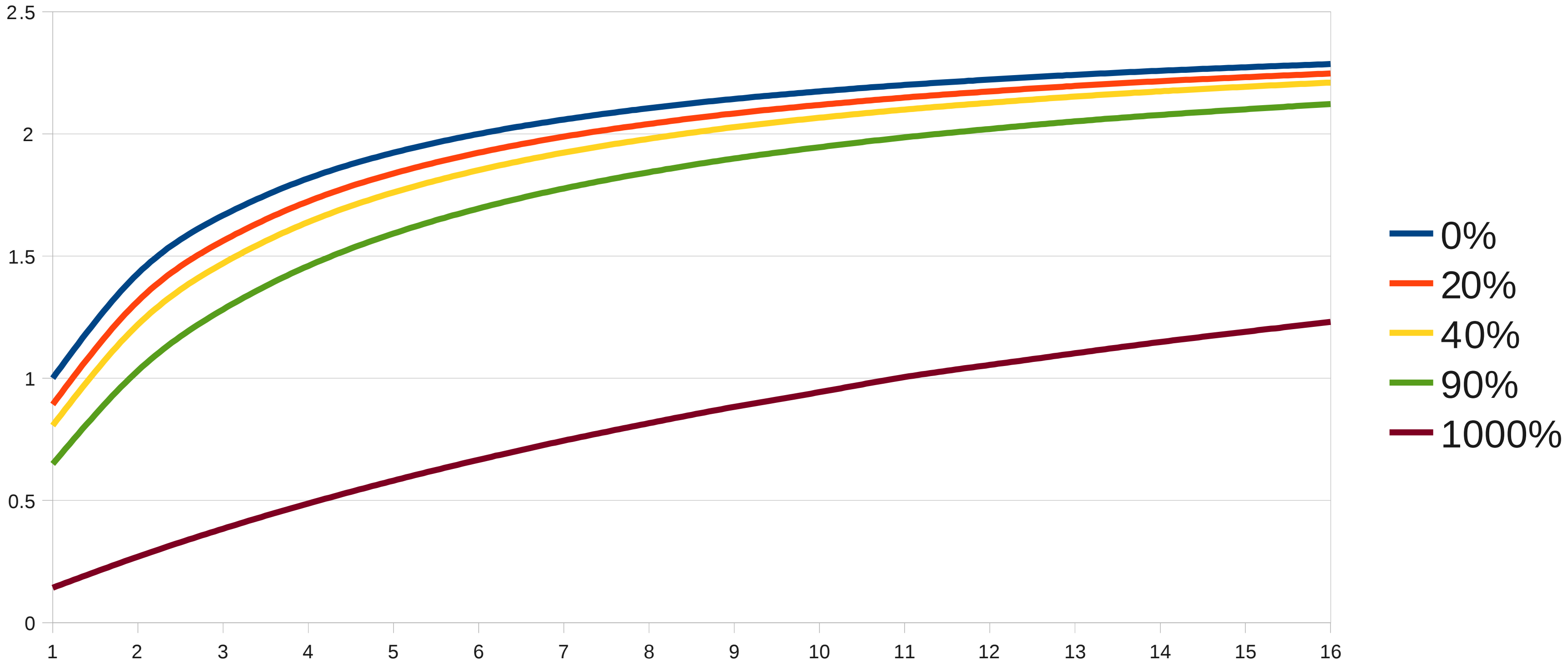
A More Realistic Formula

- Extended Amdahl's Law with Overhead

$$S = \frac{1}{(1-P)\,(1+O_S) + \frac{P}{N}\,(1+O_P)}$$

- Parallelization is not free
  - Most of the time not even for serial code
- The results are not *that* bad…

Even with Overhead P=0.6



- Even with 40% overhead not that much slower
- Speed-up from two threads on
  - Eleven threads for 10x slowdown

Programming Goals

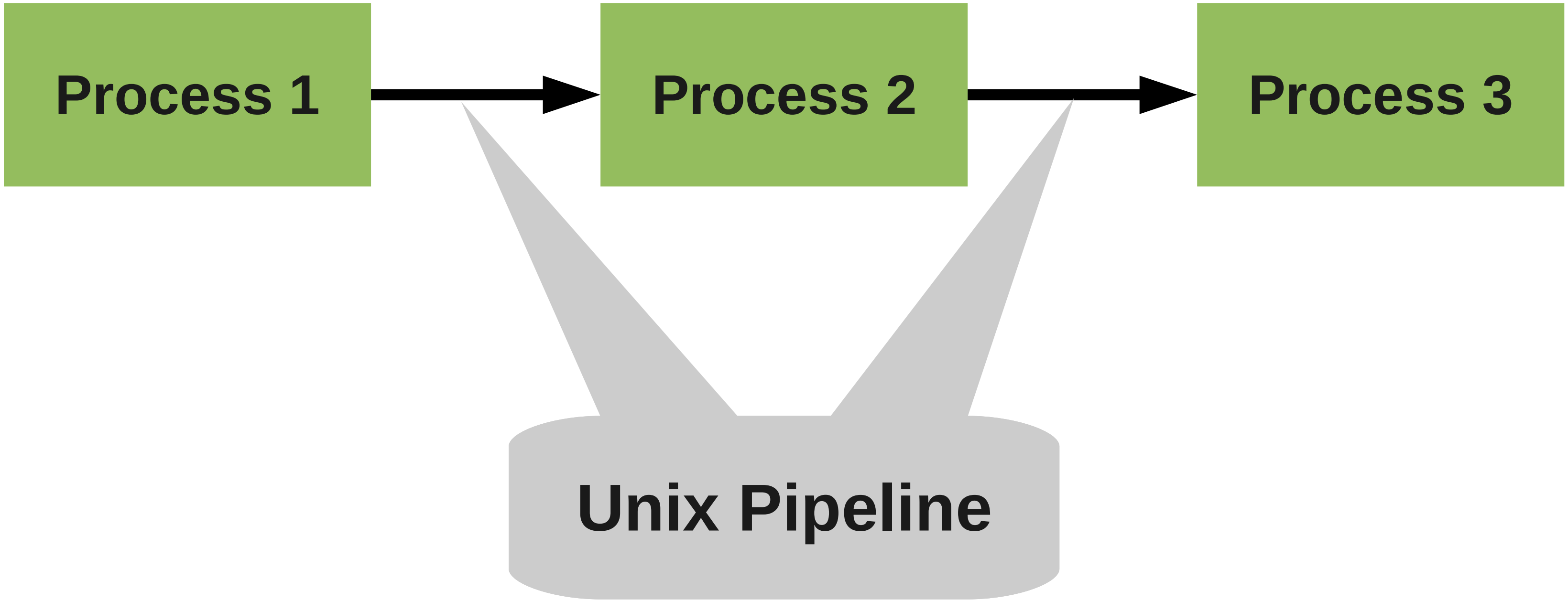
$$S = \frac{1}{(1-P)\,(1+O_S) + \frac{P}{N}\,(1+O_P)}$$

- Two goals: 1. ease parallel programming to increase P

  2. reduce O_S and O_P

Getting Parallelism

- Multi-process Pipeline



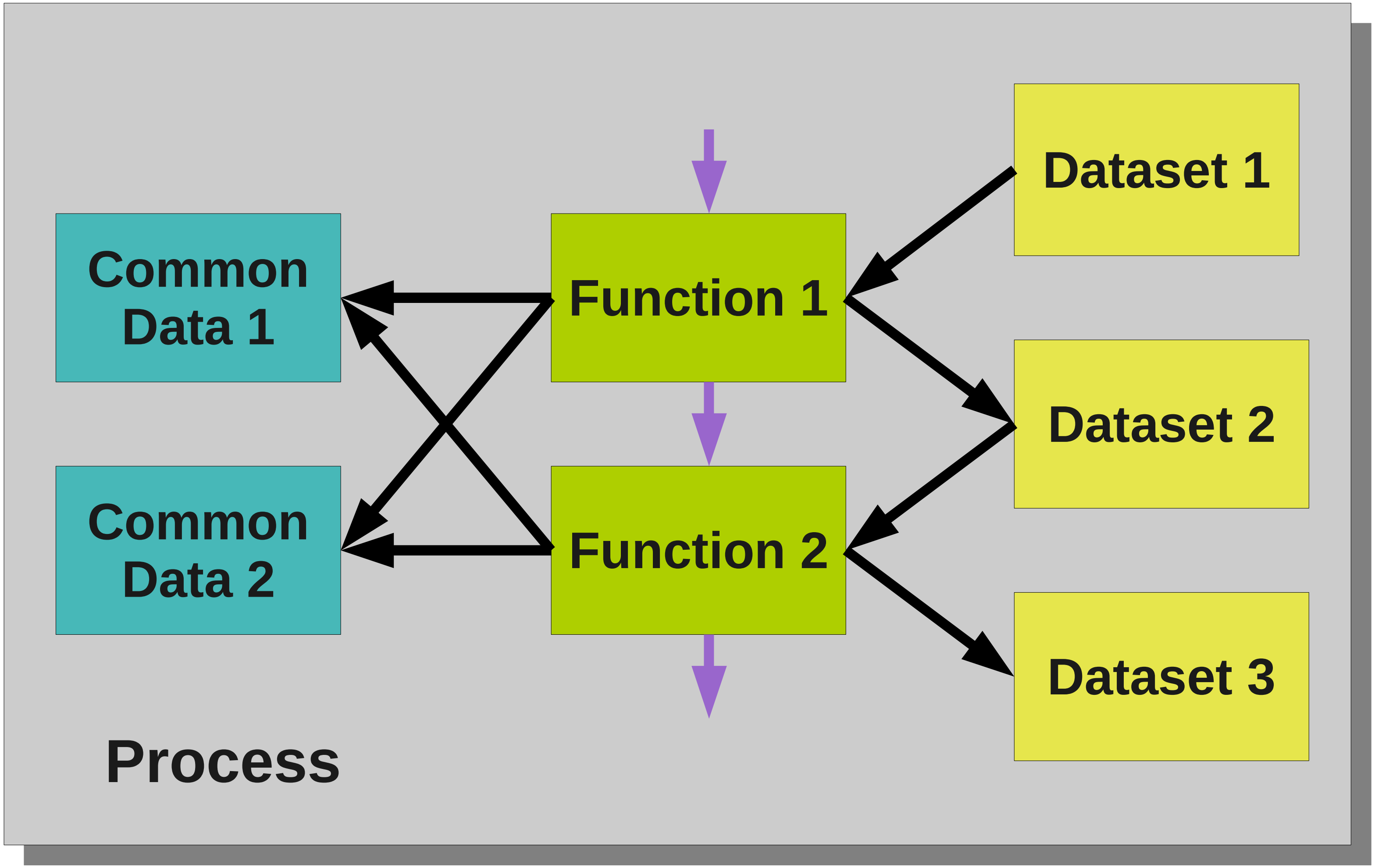
Problems with Pipelines

- Marshalling needed for transmission
- Protocol standardization required
- Limited buffer sizes
  - Lots of scheduling needed
- Program need to be designed for pipeline
  - Extending an existing program not easy
  - Major code restructuring needed

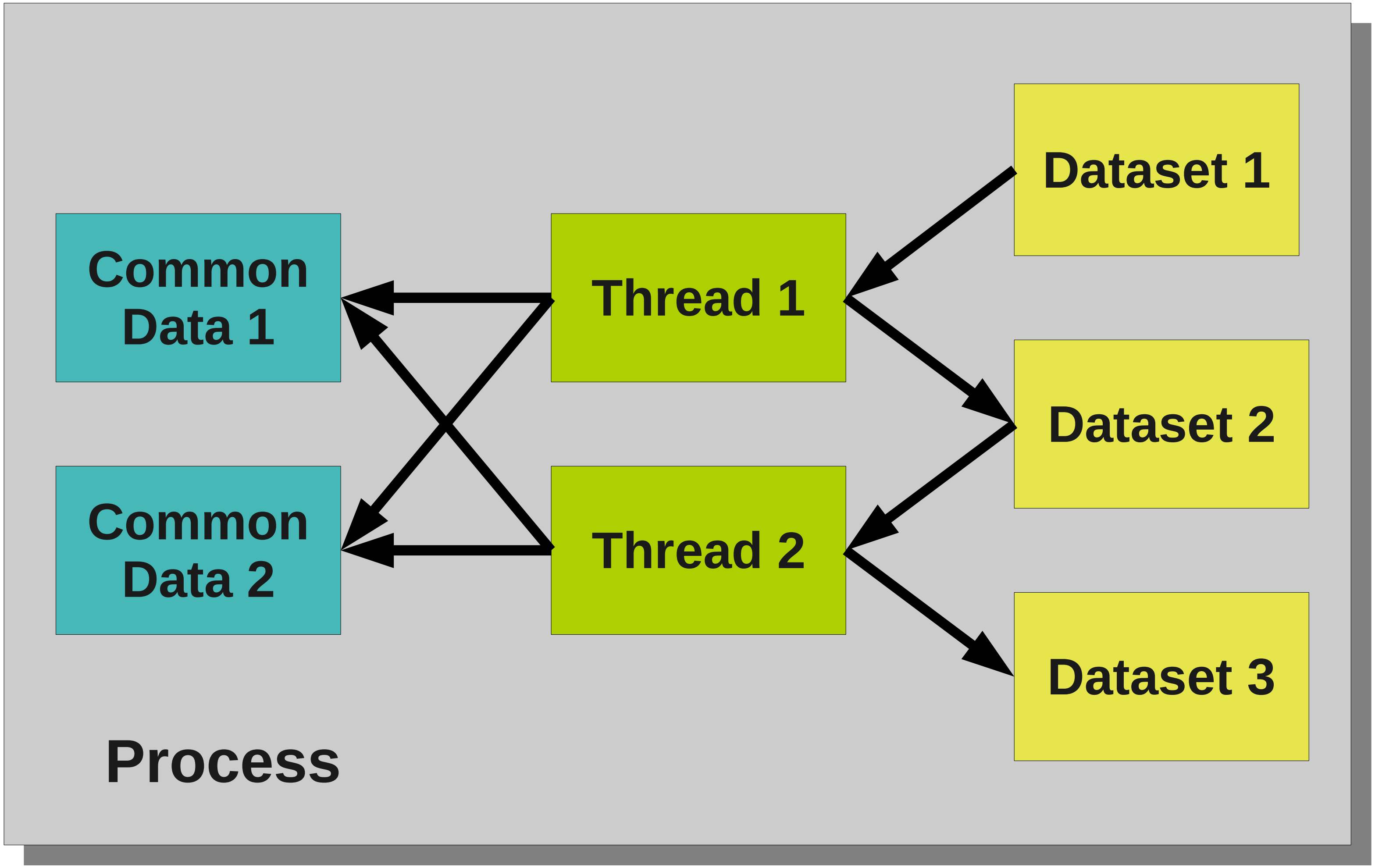
Problems with Pipelines

- Marshalling needed for transmission
- Protocol standardization required
- Limited buffer sizes
  - Lots of scheduling needed
- Program need to be designed for pipeline
  - Extending an existing program not easy
  - **Major code restructuring needed**

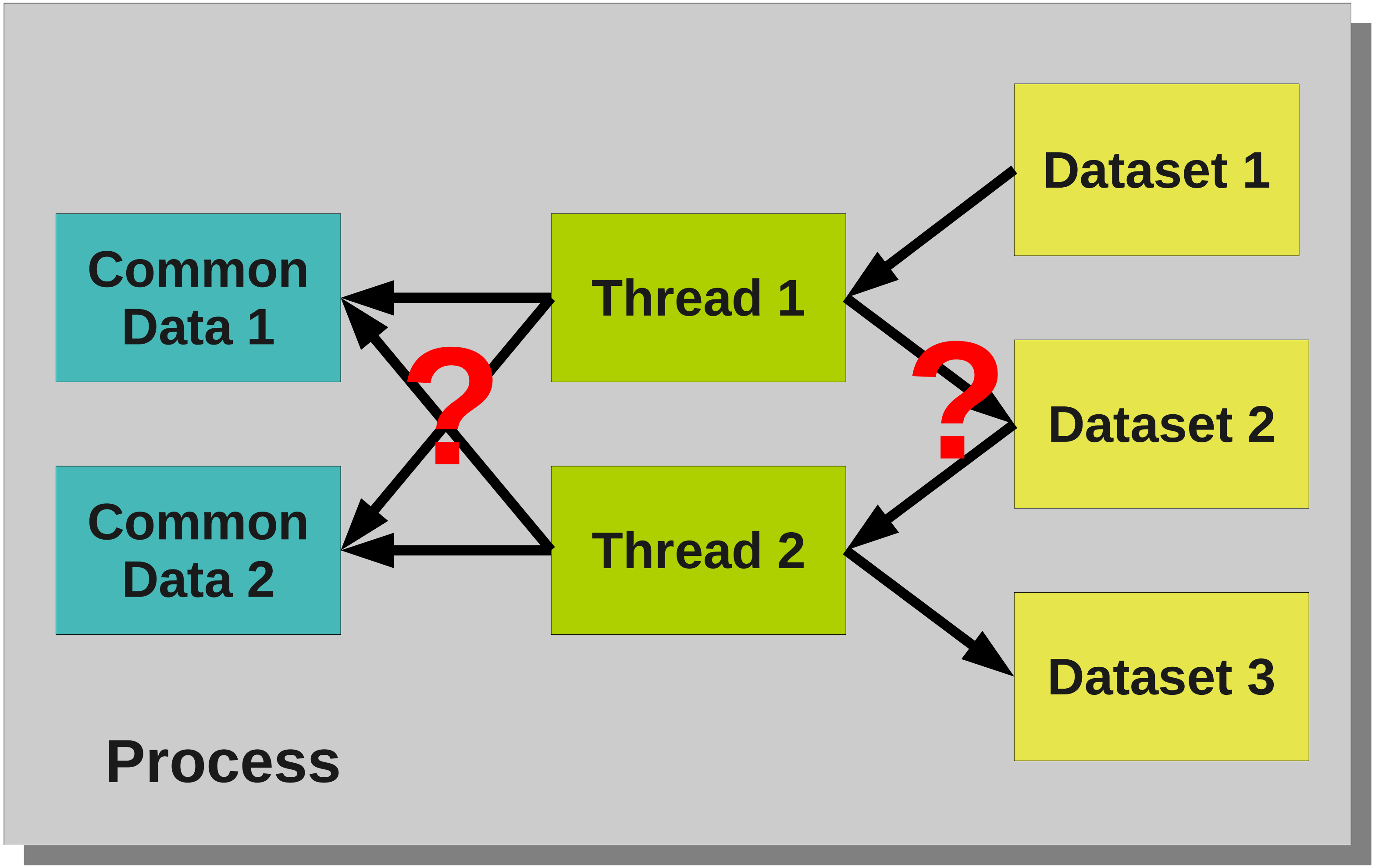
Simple Program Structure



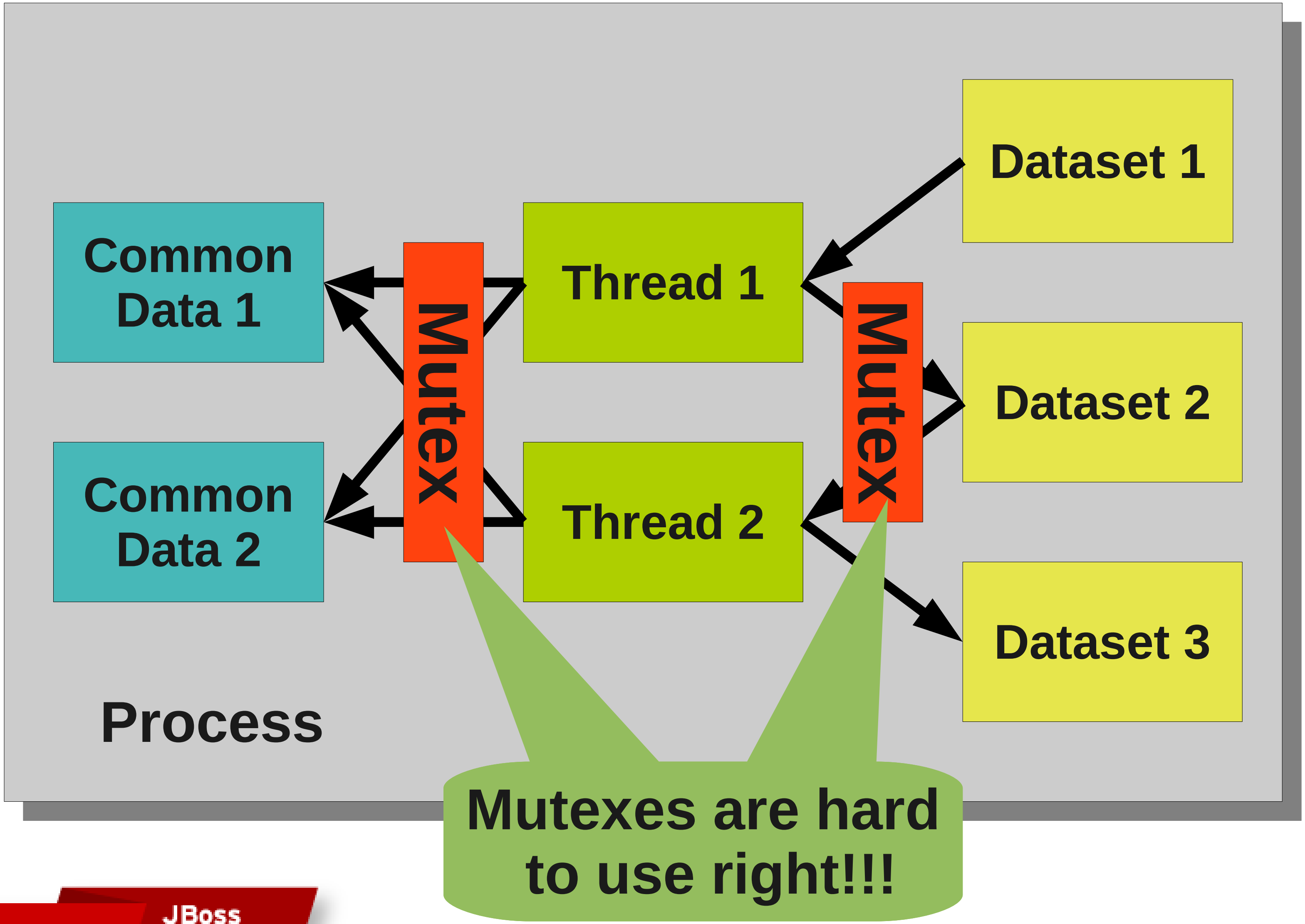
“Easy” Fix


Dataset 1
Common Data 1
Thread 1
Dataset 2
Common Data 2
Thread 2
Dataset 3
Process

“Easy” Fix


Common Data 1
Common Data 2
Thread 1
Thread 2
Dataset 1
Dataset 2
Dataset 3
?
?
Process

It seems easy…



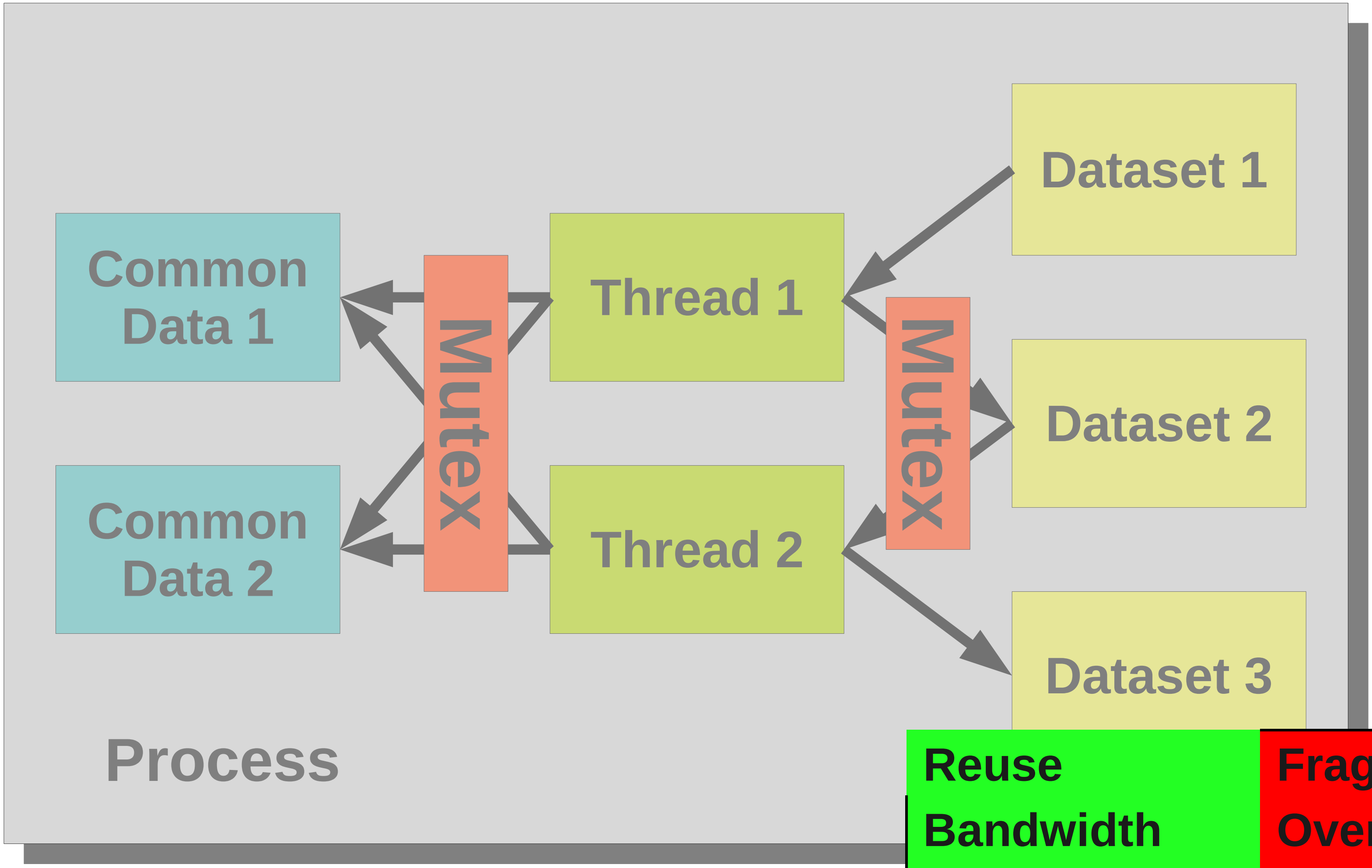
Mutexes are hard to use right!!!

Explicit Multi-Threading

- Ill-conceived solution
  - Yes
    - Existing code can be reused, easier to set up
    - High-bandwidth inter-thread communication
    - On some OSes context switching faster
  - But:
    - Fragile programming model (one thread dies, the process dies)
    - Memory handling mistakes have global effects
    - Unix model initially not designed for multiple threads

Hard to write correct code! High Cost!

Measures



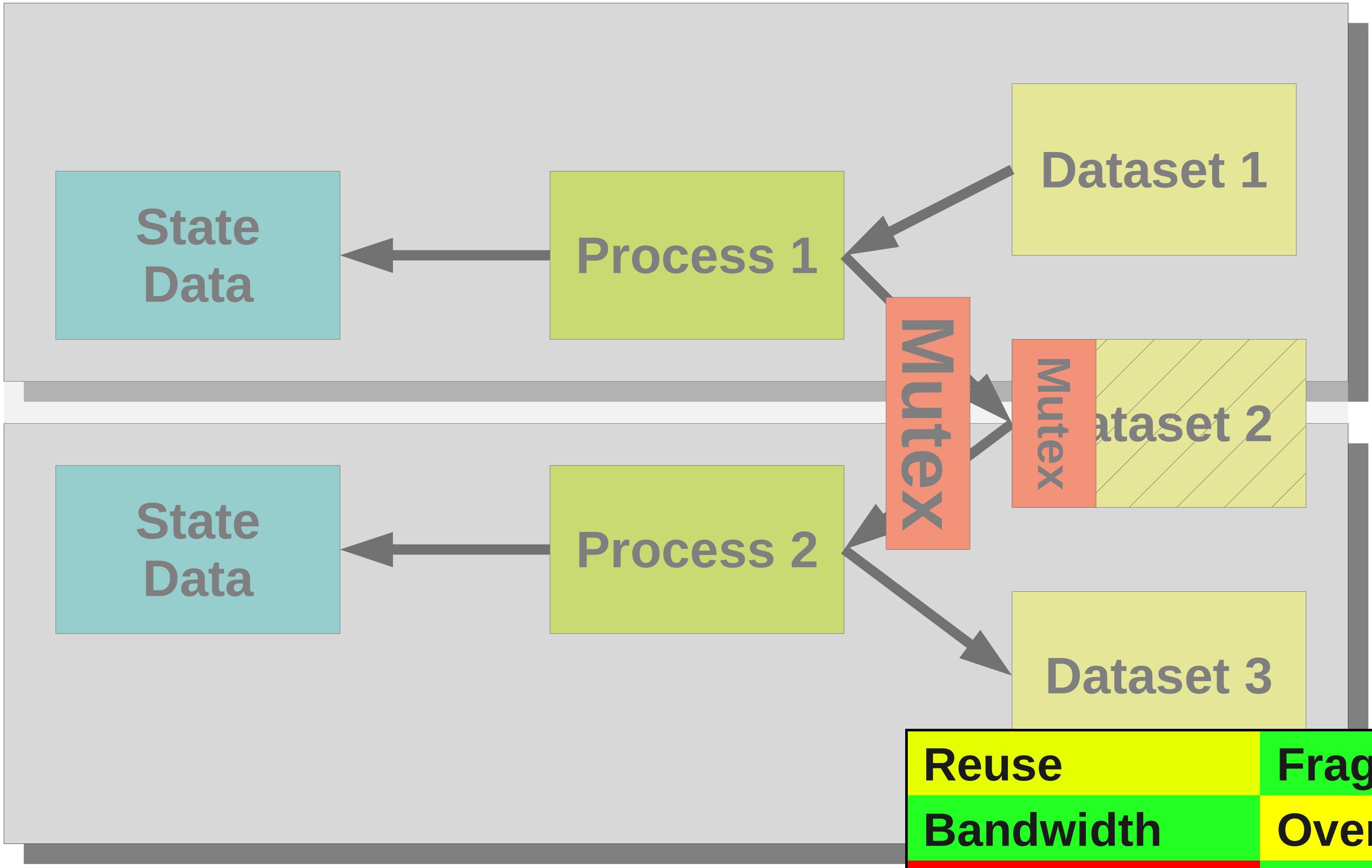
Reuse	Fragile
Bandwidth	Overwrites
Context Cost	Unix model
Ease Program	Error Prone

Alternative 1: fork and Shared Memory

- All in POSIX:

```
int fd = shm_open(name, O_RDWR|O_CREAT);
ftruncate(fd, size);
p = mmap(NULL, size, PROT_READ|PROT_WRITE,
         MAP_SHARED, fd, 0);
if (fork() == 0)
   ...
```

`fork` and Shared Memory

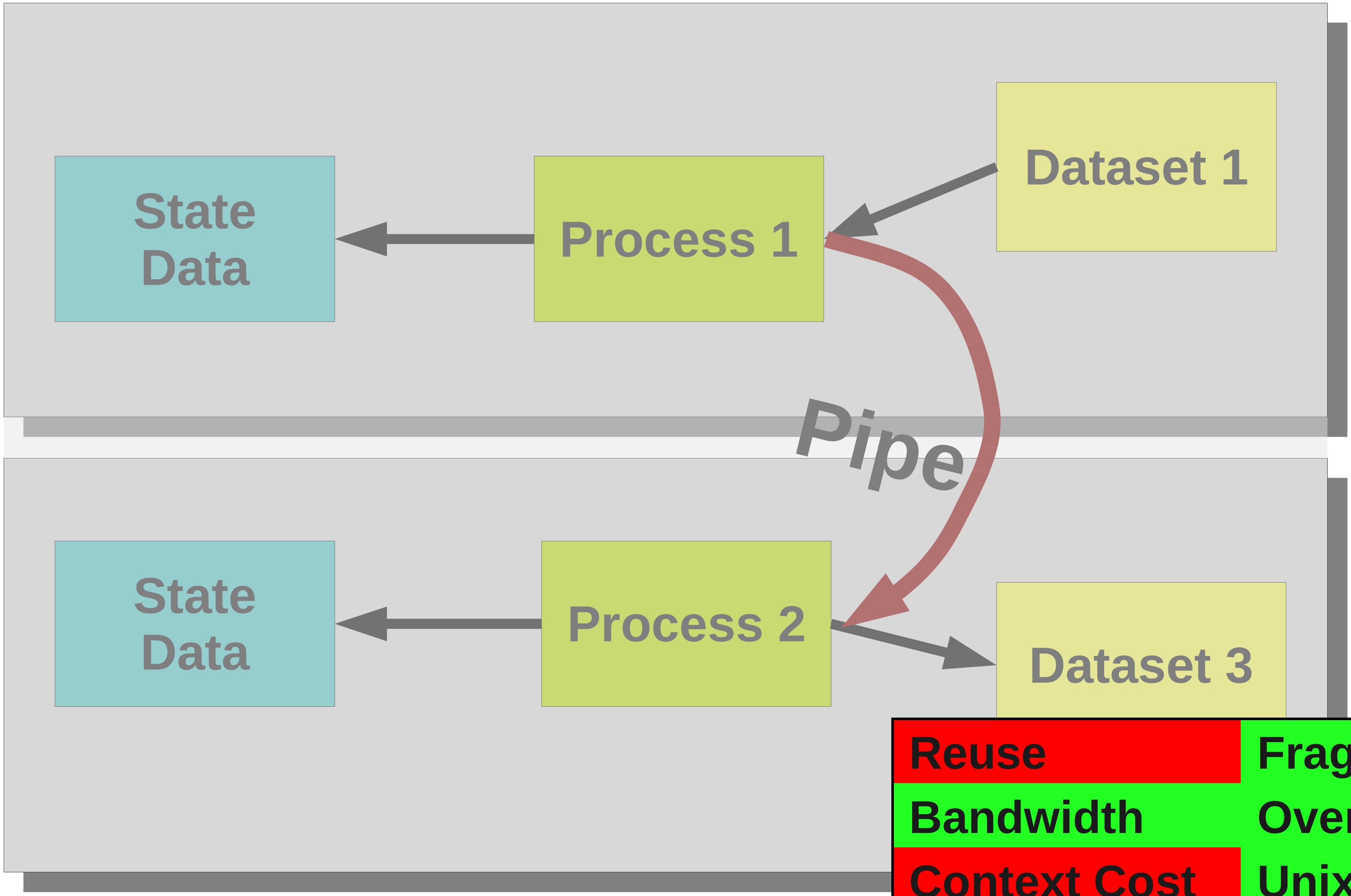


Reuse	Fragile
Bandwidth	Overwrites
Context Cost	Unix model
Ease Program	Error Prone

Alternative 2: `fork` and Linux Pipes

- Linux extensions, not POSIX (yet ☺)
- Can be zero-copy
- Use if just transferring data without inspection
- splice: transfer from file descriptor to pipe
- tee: transfer between pipes and keep data usable
- vmsplice: transfer from memory to pipe

fork and Linux Pipes



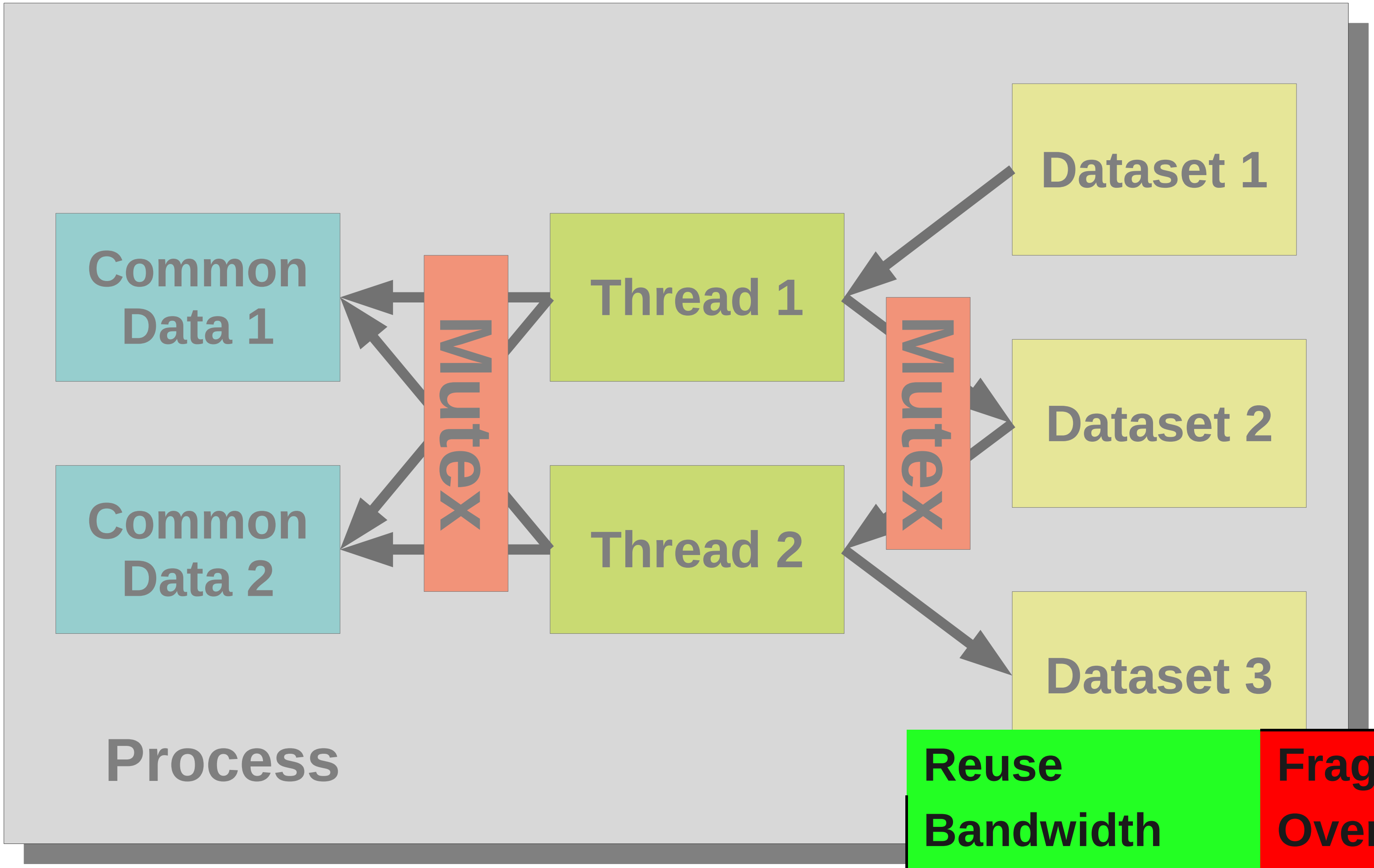
Reuse	Fragile
Bandwidth	Overwrites
Context Cost	Unix model
Ease Program	Error Prone

Alternative 3: Thread Local Storage

- Use thread-local storage
  - Very much simplifies use of static variables
  - No more false sharing of cache lines

```
__thread struct foo var;
```

Thread Local Storage

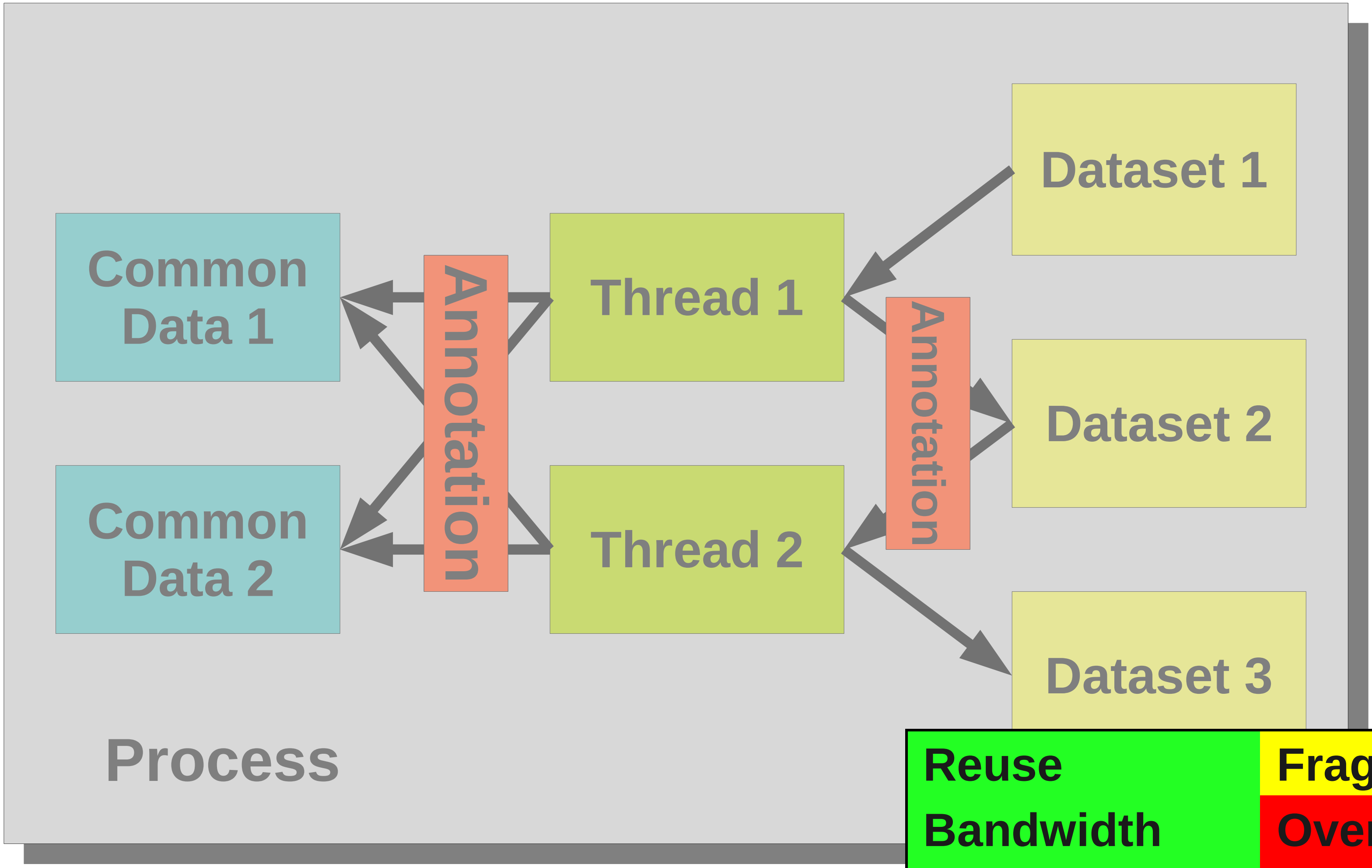


Reuse	Fragile
Bandwidth	Overwrites
Context Cost	Unix model
Ease Program	Error Prone

Alternative 4: OpenMP

- Language extension to C, C++, Fortran languages
- Implements many thread functions with very simple interface for
  - Thread creation (controlled)
  - Exclusion
  - Thread-local Data

OpenMP

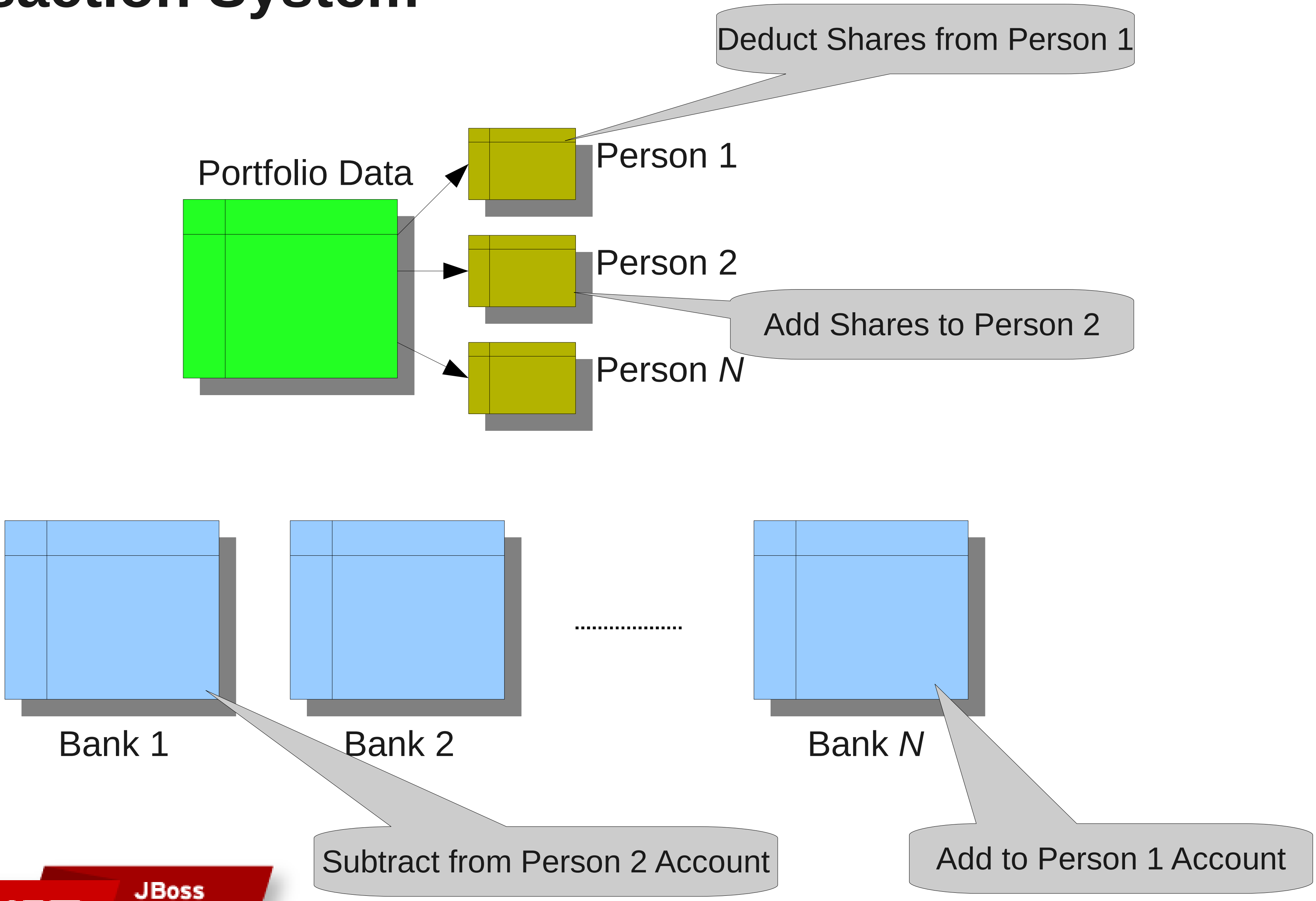


Reuse	Fragile
Bandwidth	Overwrites
Context Cost	Unix model
Ease Program	Error Prone

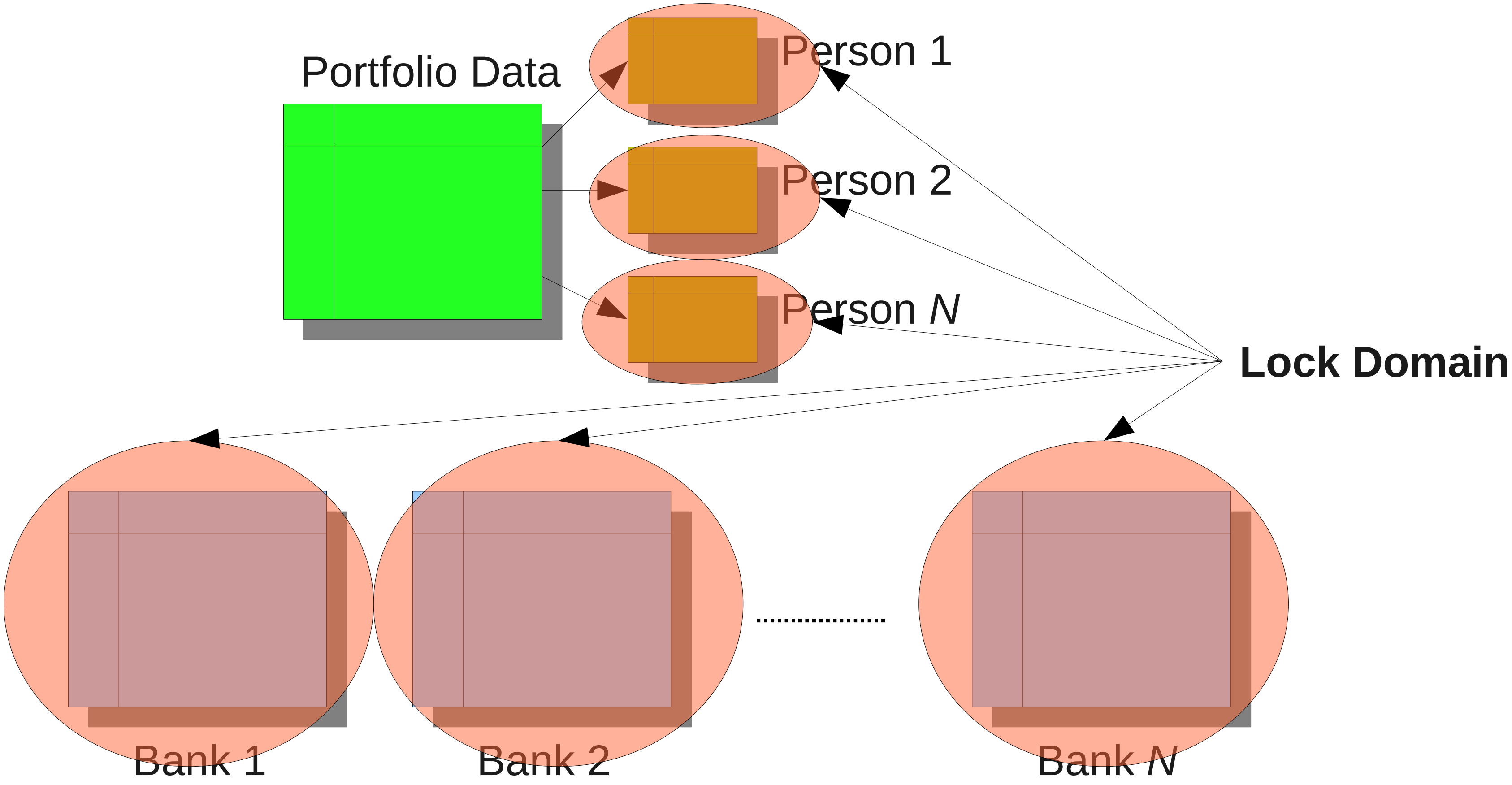
Alternative 5: Transactional Memory

- Extensions to C and C++ languages
- Can help to avoid using mutexes
  - Just source code annotations
  - No more deadlocks!!
  - Fine-grained locking without the problems
- Slow as pure software solutions
  - Hardware support on the horizon

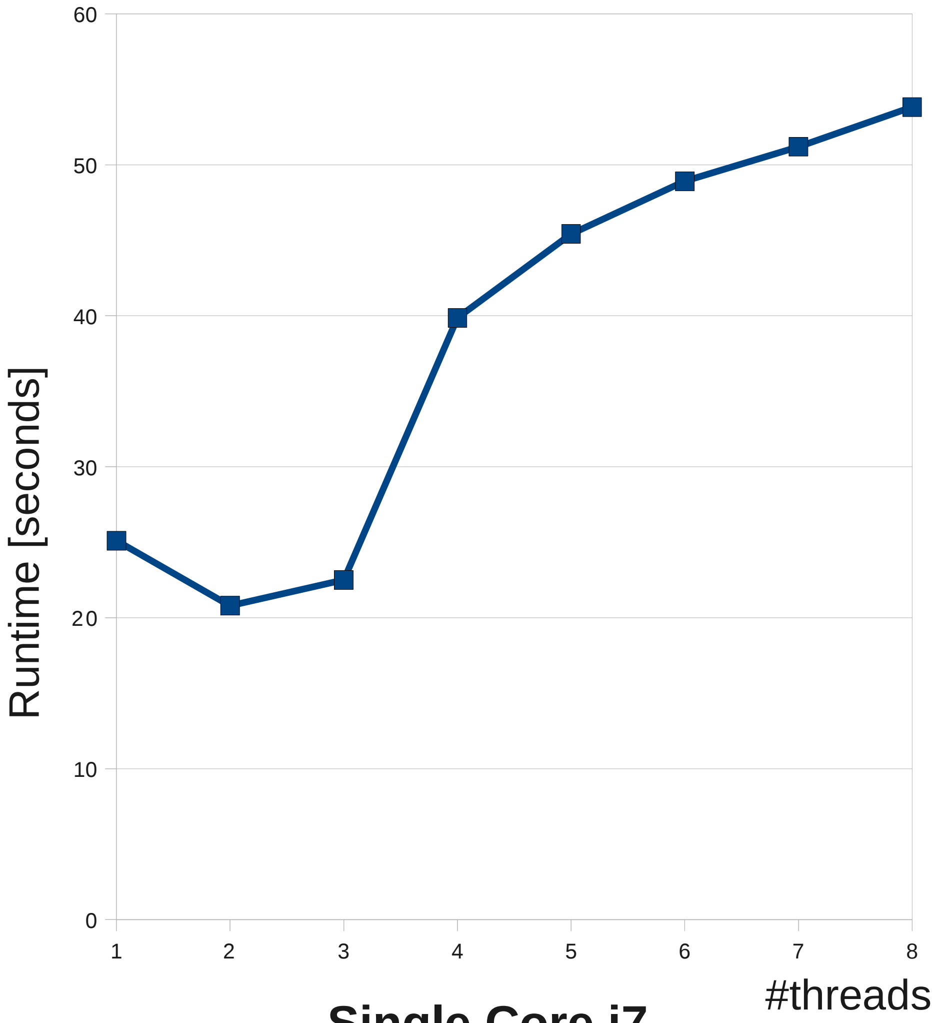
Transaction System



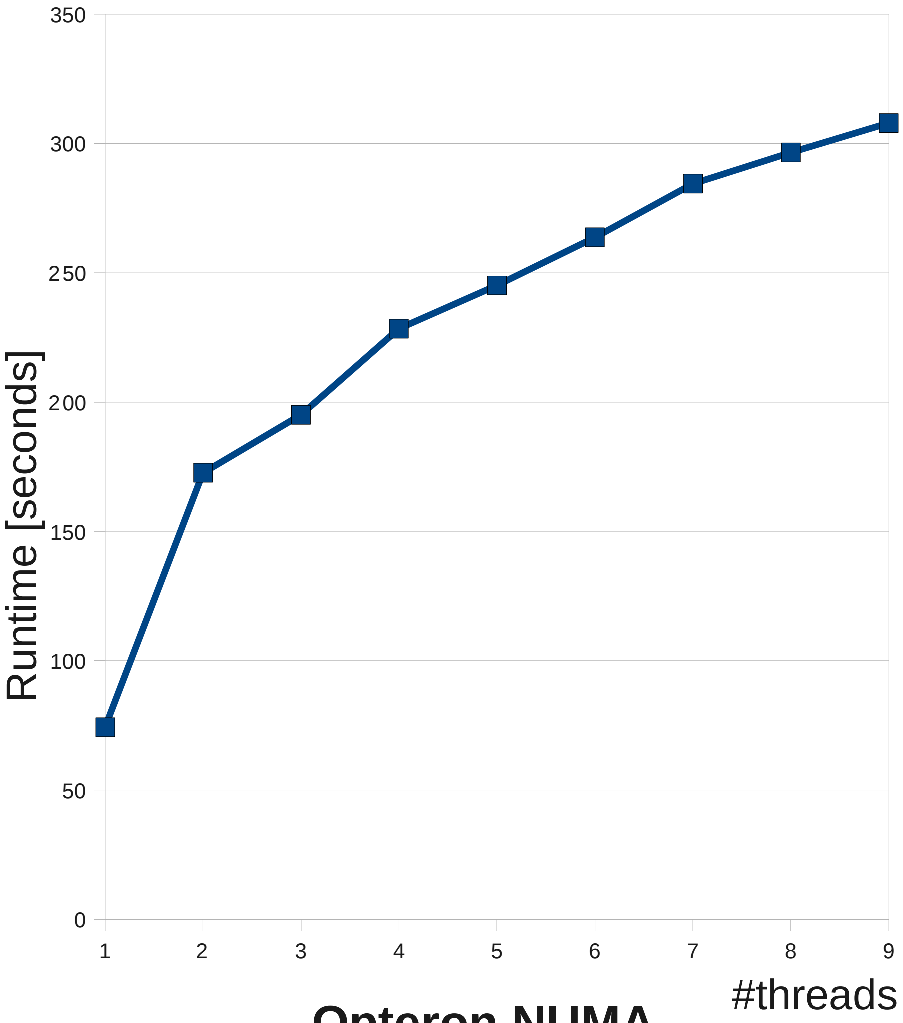
Trying to Parallelize



Not What We Want

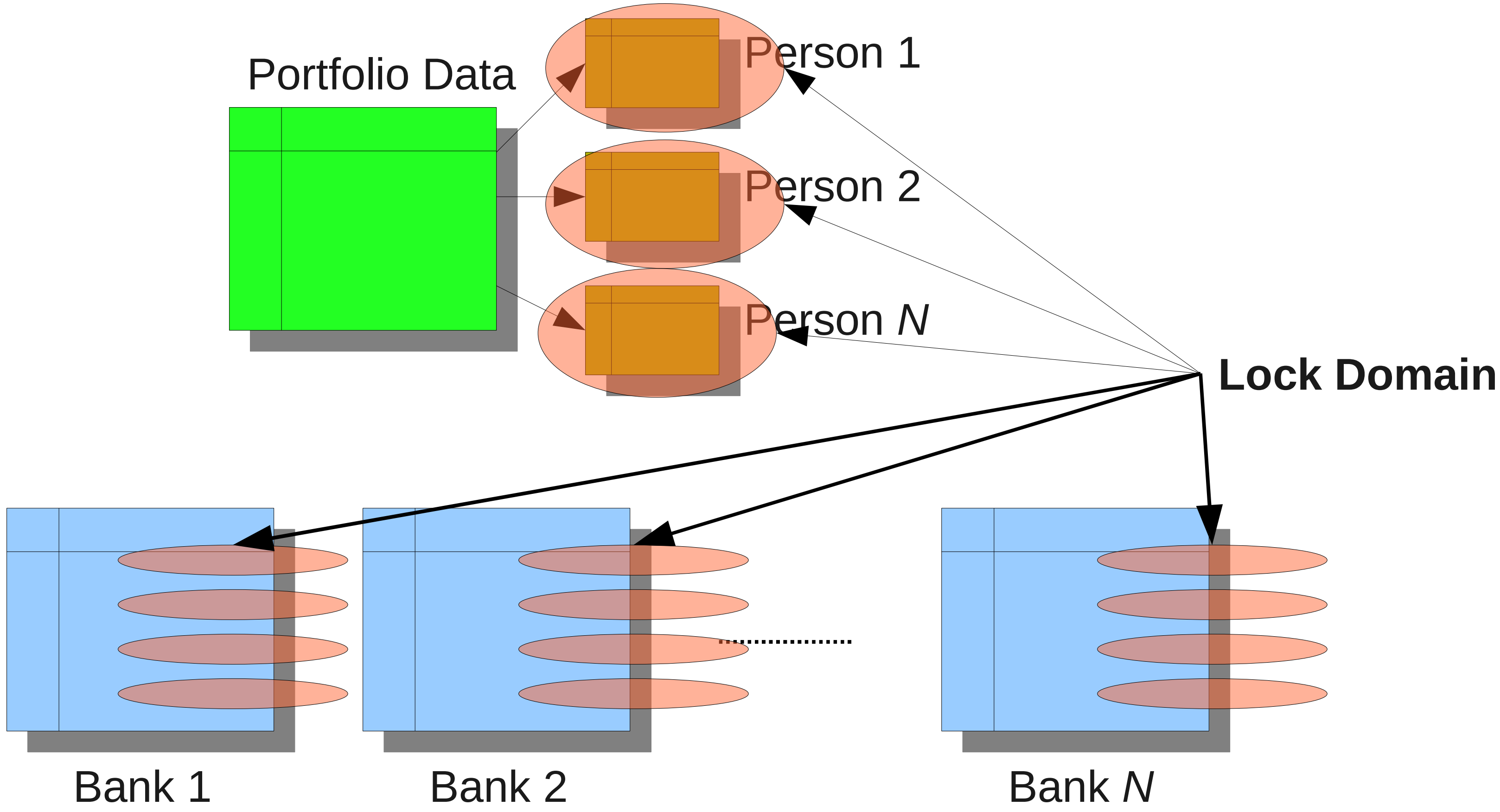


Single Core i7

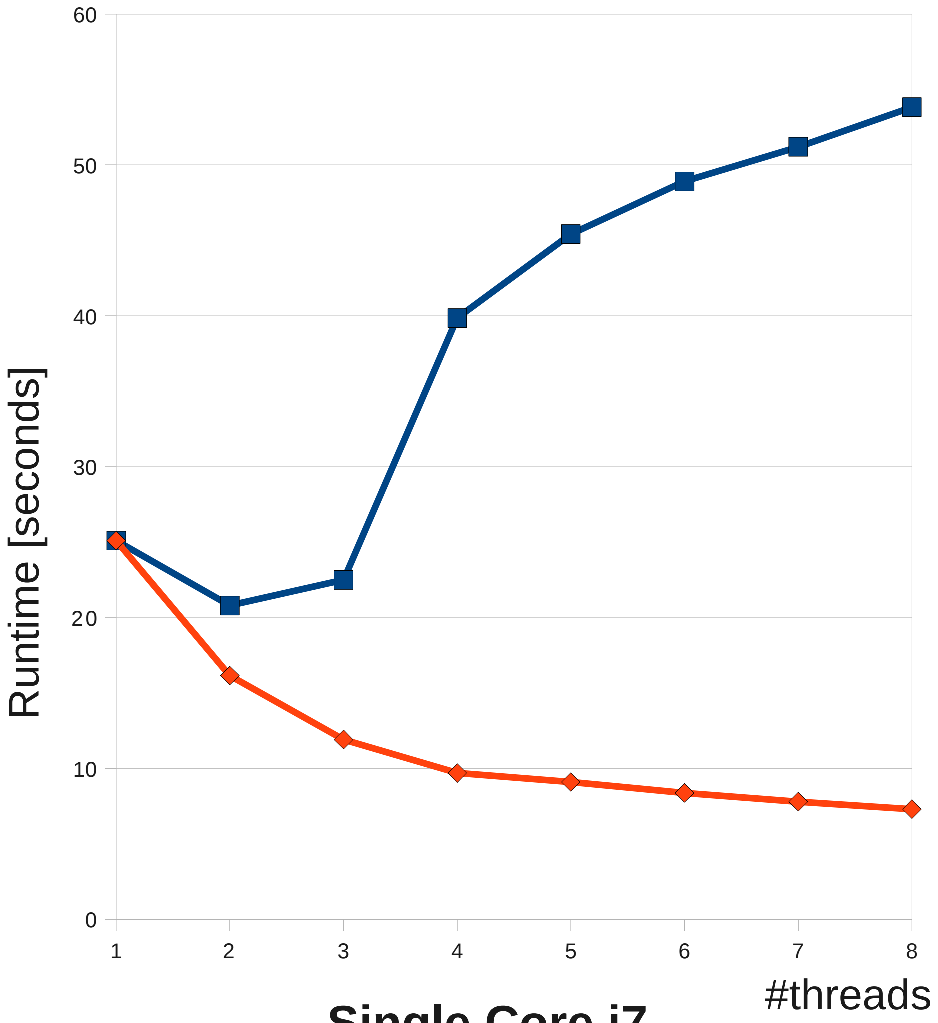


Opteron NUMA

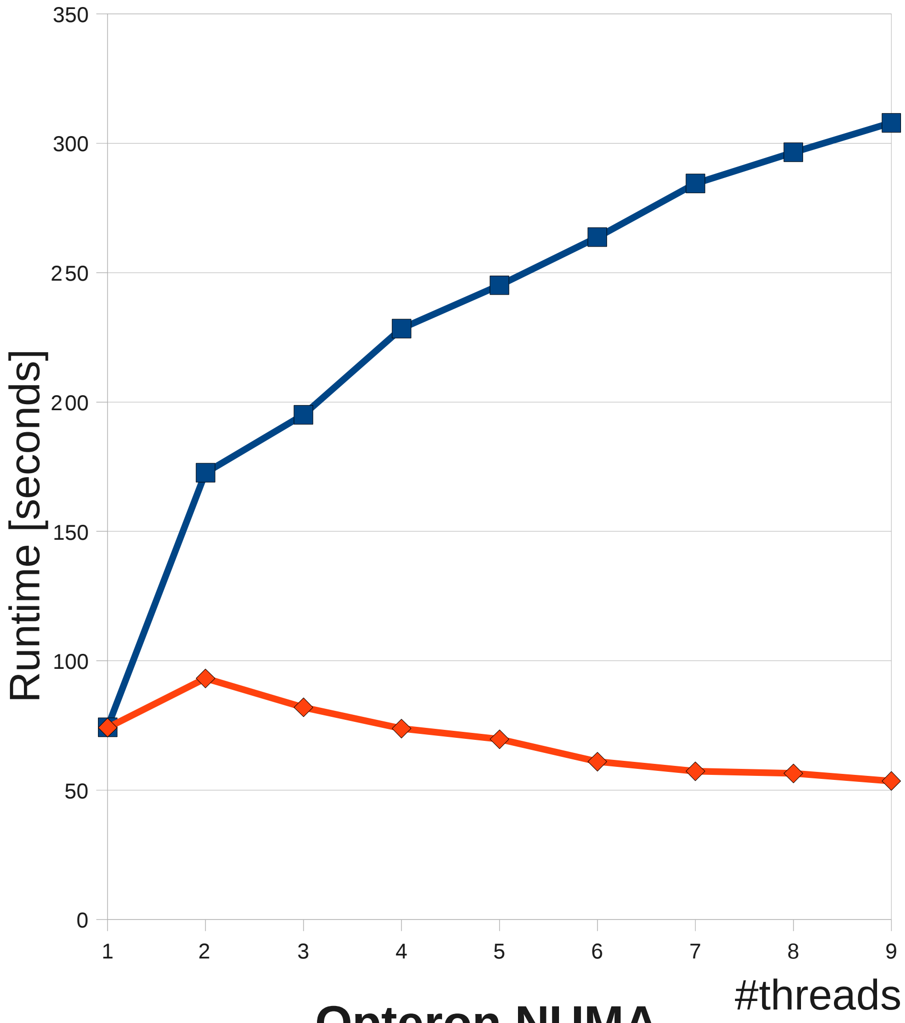
Trying to Parallelize



Somewhat Better But…

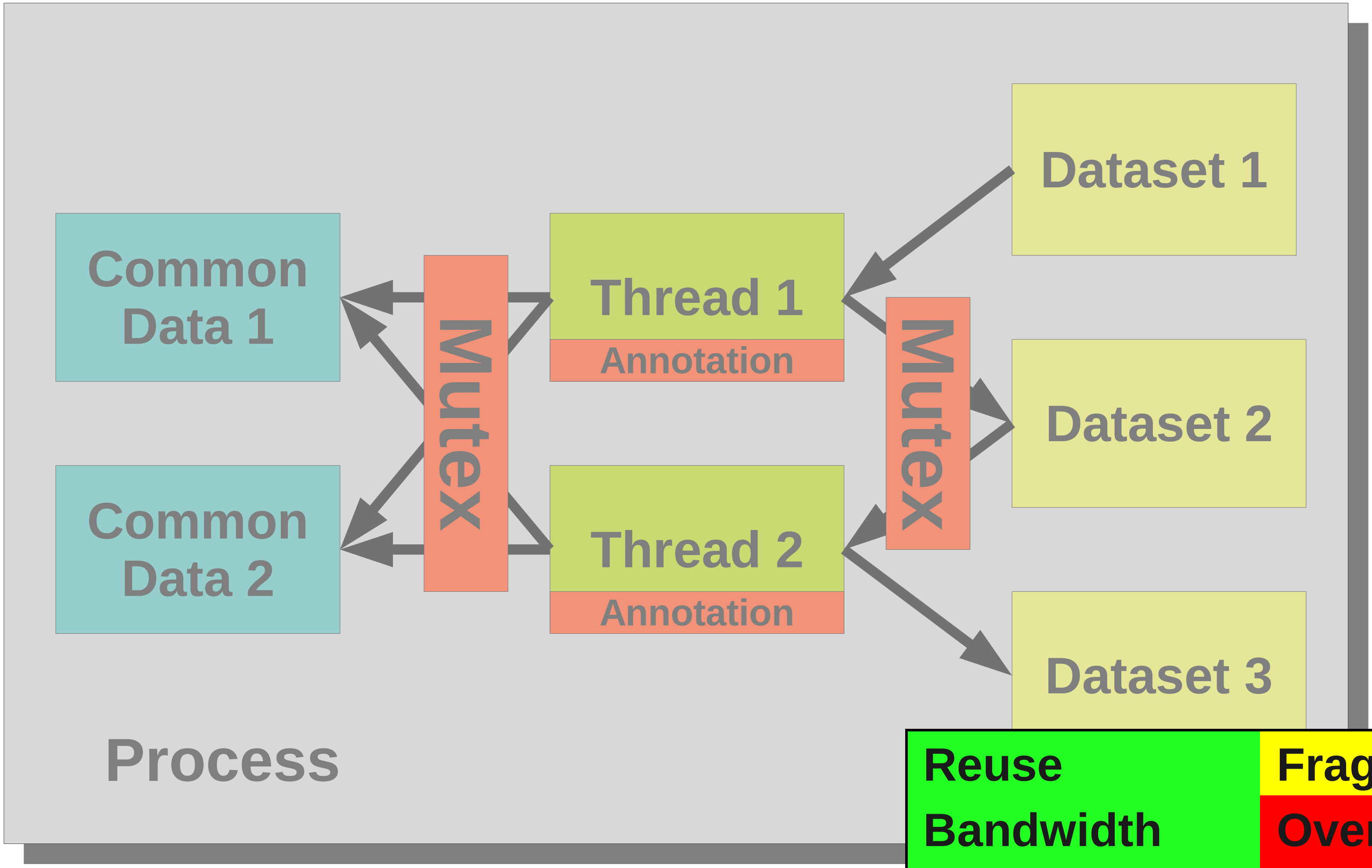


Single Core i7



Opteron NUMA

Transactional Memory



Reuse	Fragile
Bandwidth	Overwrites
Context Cost	Unix model
Ease Program	Error Prone

Conclusion

- Abilities to exploit hardware are there
  - Explicit threading only for experts
- But there is a lot of help
  - Use processes, not threads; or
  - If threads are used combine
    - Thread-local storage
    - Implicit thread creation
      - OpenMP
      - Futures
    - Transactional memory

Questions?

FOLLOW US ON TWITTER

www.twitter.com/redhatsummit

TWEET ABOUT IT

#summitjbw

READ THE BLOG

http://summitblog.redhat.com/