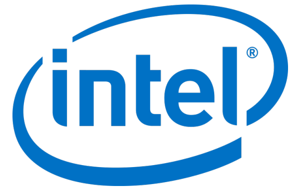


Parallel Programming with Lisp for Performance

Pascal Costanza, Intel, Belgium
European Lisp Symposium 2014, Paris, France
The views expressed are my own, and not those of my employer.

Legal Notices

- Cilk, Intel, the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries. Other names and brands may be claimed as the property of others. Results have been estimated based on internal Intel analysis and are provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance. Intel does not control or audit the design or implementation of third party benchmark data or Web sites referenced in this document. Intel encourages all of its customers to visit the referenced Web sites or others where similar performance benchmark data are reported and confirm whether the referenced benchmark data are accurate and reflect performance of systems available for purchase.

- Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instructions sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. (Notice revision #20110804)

- Copyright © 2014 Intel Corporation. All rights reserved.

Introduction

- SMP has arrived in Common Lisp implementations (ABCL, Allegro, clisp, Clozure, ECL, LispWorks, SBCL, …)
- Parallel programming means not only starting lots of threads, but also synchronizing them.
- So far LispWorks seems to provide the richest library of synchronization mechanisms. Therefore, in this presentation, the focus is on LispWorks.

Why is parallel programming hard?

- Consider this simple program:

```
(defun count-3s (list)
  (let ((count 0))
    (dolist (number list)
      (mp:process-run-function "count numbers" '()
        (lambda () (when (= number 3) (incf count)))))
    count))
```

Why is parallel programming hard?

- Consider this simple program:

```
(defun count-3s (list)
  (let ((count 0))
    (dolist (number list)
      (mp:process-run-function "count numbers" '()
        (lambda () (when (= number 3) (incf count)))))
    count))
```

Why is parallel programming hard?

- Consider this simple program:

```
(defun count-3s (list)
  (let ((count 0))
    (dolist (number list)
      (mp:process-run-function "count numbers" '()
        (lambda () (when (= number 3) (incf count)))))
    count))
```

Why is parallel programming hard?

- Consider this simple program:

```
(defun count-3s (list)
   (let ((count 0))
      (dolist (number list)
         (mp:process-run-function "count numbers" '()
            (lambda () (when (= number 3) (incf count)))))
      count))
```

Why is parallel programming hard?

- Consider this simple program:

```
(defun count-3s (list)
  (let ((count 0))
    (dolist (number list)
      (prun (lambda () (when (= number 3) (incf count)))))
    count))
```

Why is parallel programming hard?

- Consider this simple program:

```
(defun count-3s (list)
  (let ((count 0))
    (dolist (number list)
      (prun (lambda () (when (= number 3) (incf count)))))
    count))
```

- Combinatorial explosion of the computational state space:
  - Accesses to the count variable can occur at the same time.
  - We do not know when all the different computations are done.
  - In Common Lisp, iterations are also not guaranteed to use fresh bindings.

Why is parallel programming hard?

- ```
  (defun count-3s (list)
    (let ((count 0)) (iteration-count 0)
          (lock (mp:make-lock)))
      (dolist (number list)
        (let ((number number))
          (prun (lambda ()
                  (mp:with-lock (lock)
                    (when (= number 3) (incf count))
                    (incf iteration-count))))))
      (loop until (= iteration-count (length list)))
      count)
  ```
  ```

Why is parallel programming hard?

```
• (defun count-3s (list)
    (let ((count 0)) (iteration-count 0)
          (lock (mp:make-lock)))
      (dolist (number list)
        (let ((number number))
          (prun (lambda ()
                    (mp:with-lock (lock)
                      (when (= number 3) (incf count))
                      (incf iteration-count))))))
      (loop until (= iteration-count (length list)))
      count)
```

Why is parallel programming hard?

- ```
  (defun count-3s (list)
    (let ((count 0)) (iteration-count 0)
          (lock (mp:make-lock)))
      (dolist (number list)
        (let ((number number))
          (prun (lambda ()
                    (mp:with-lock (lock)
                      (when (= number 3) (incf count))
                      (incf iteration-count)))))))
      (loop until (= iteration-count (length list)))
      count)
  ```

  ```

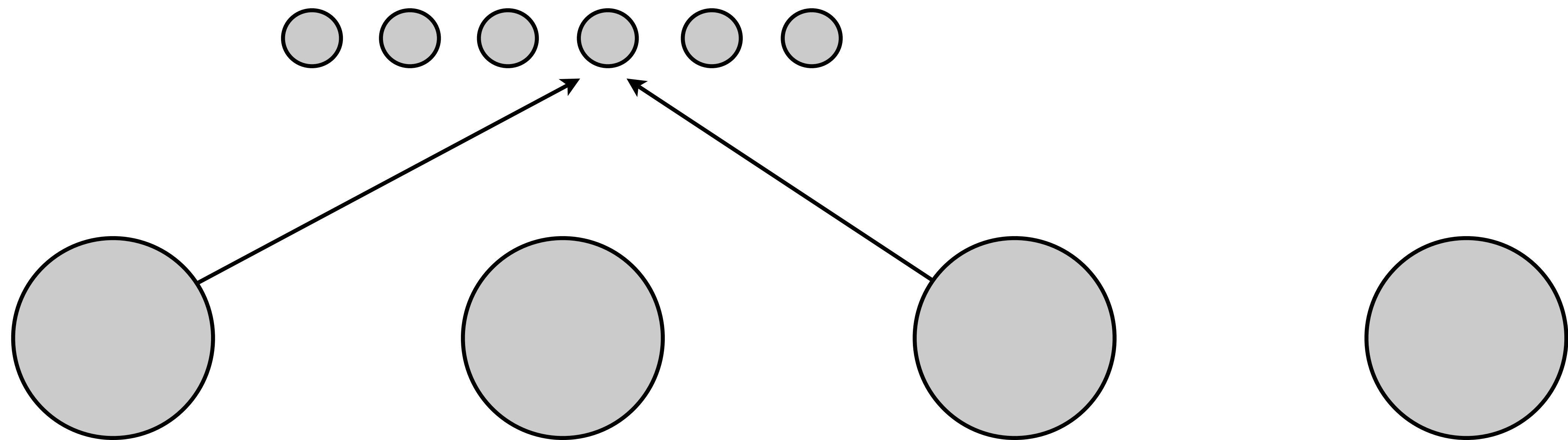
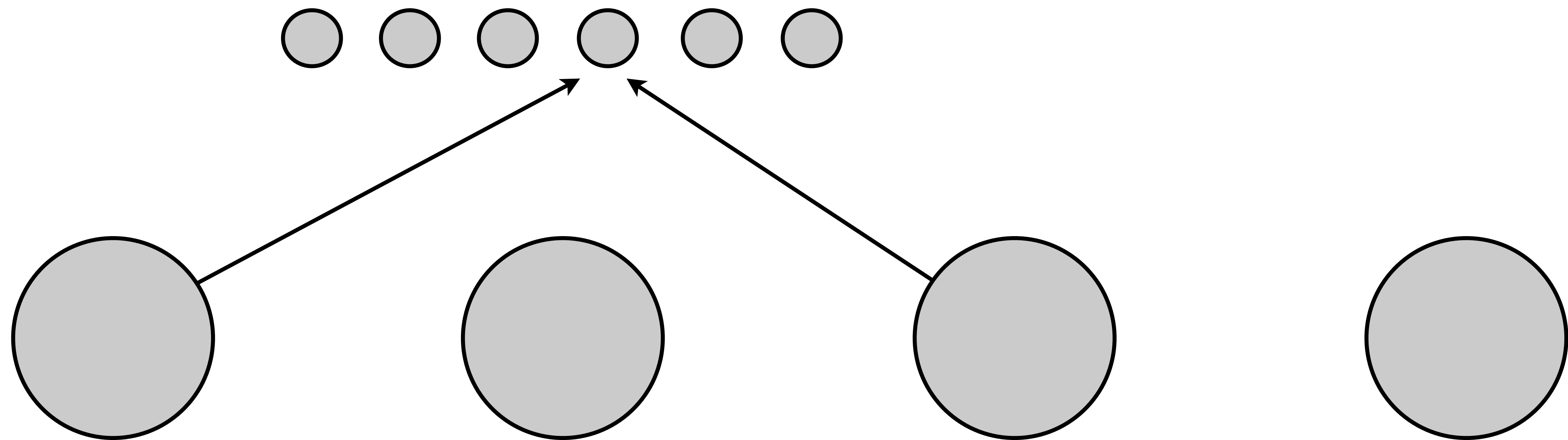
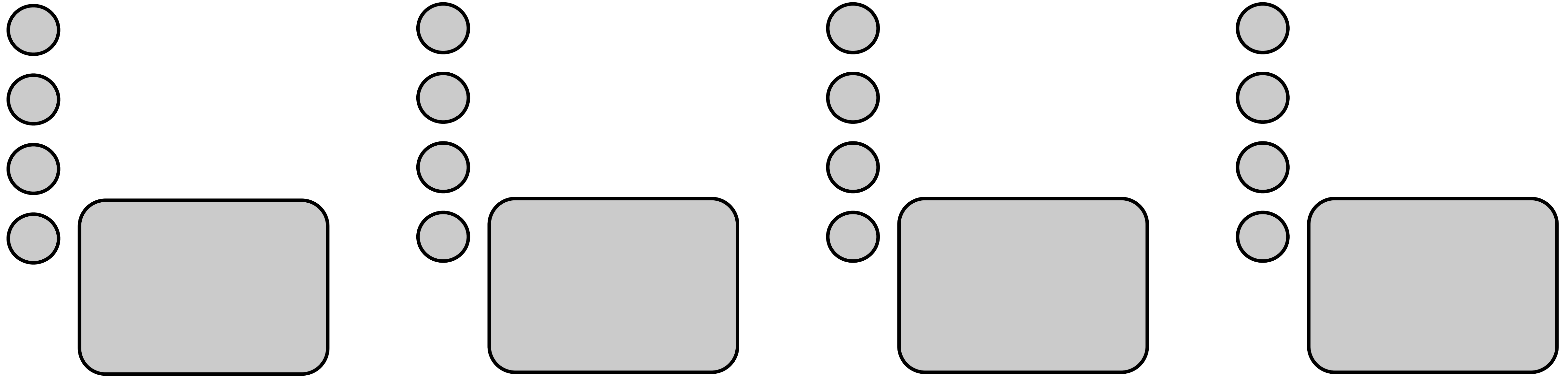
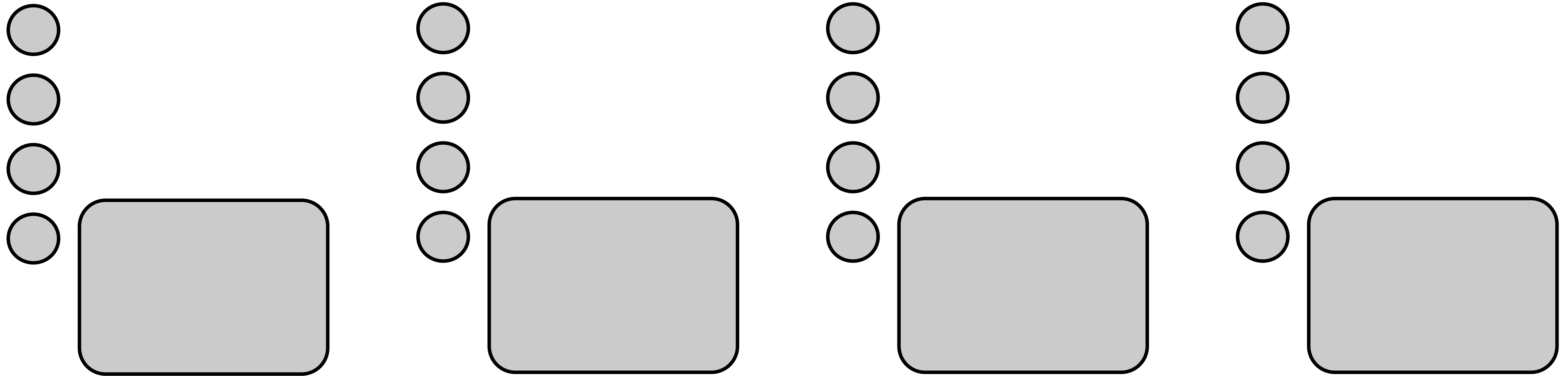
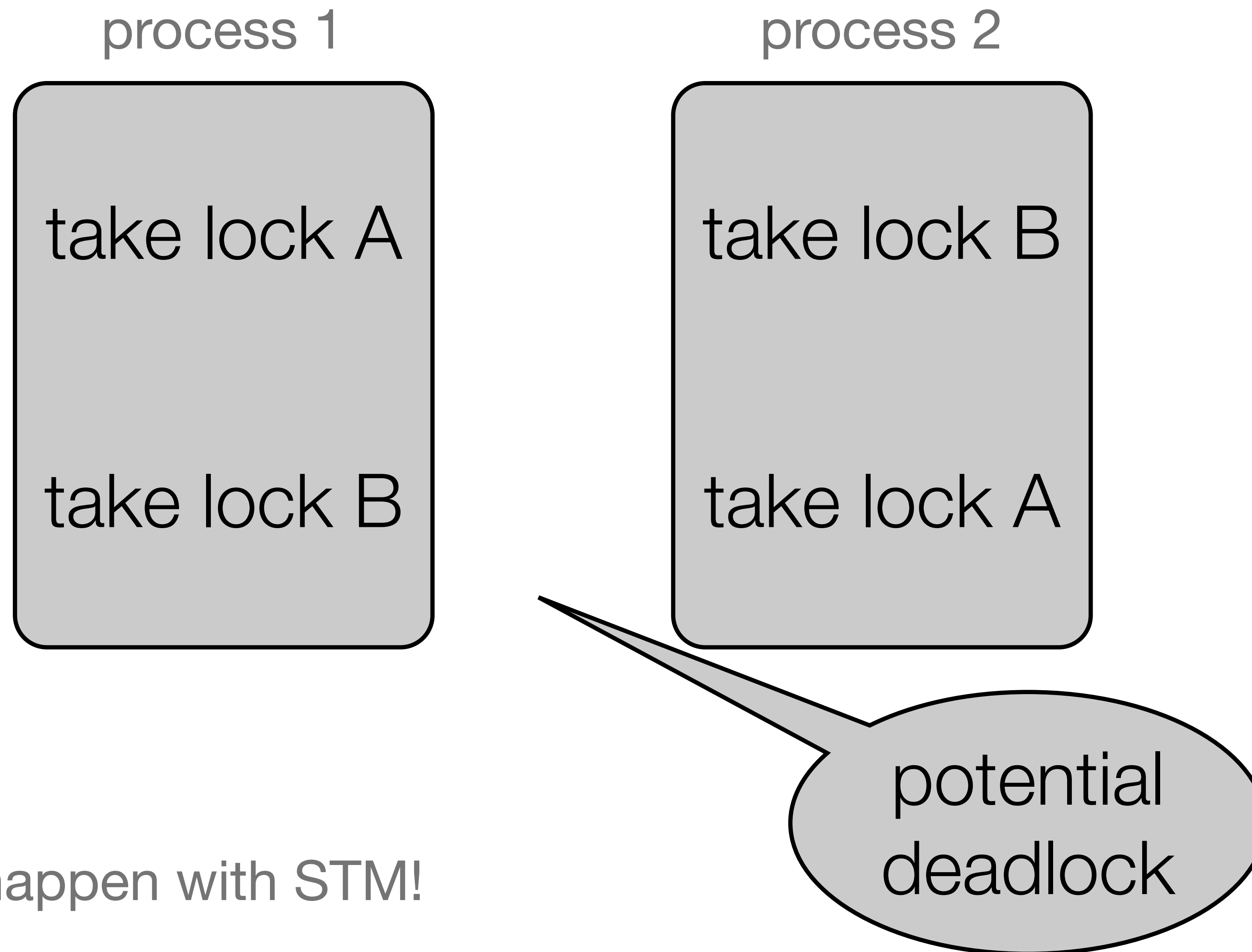
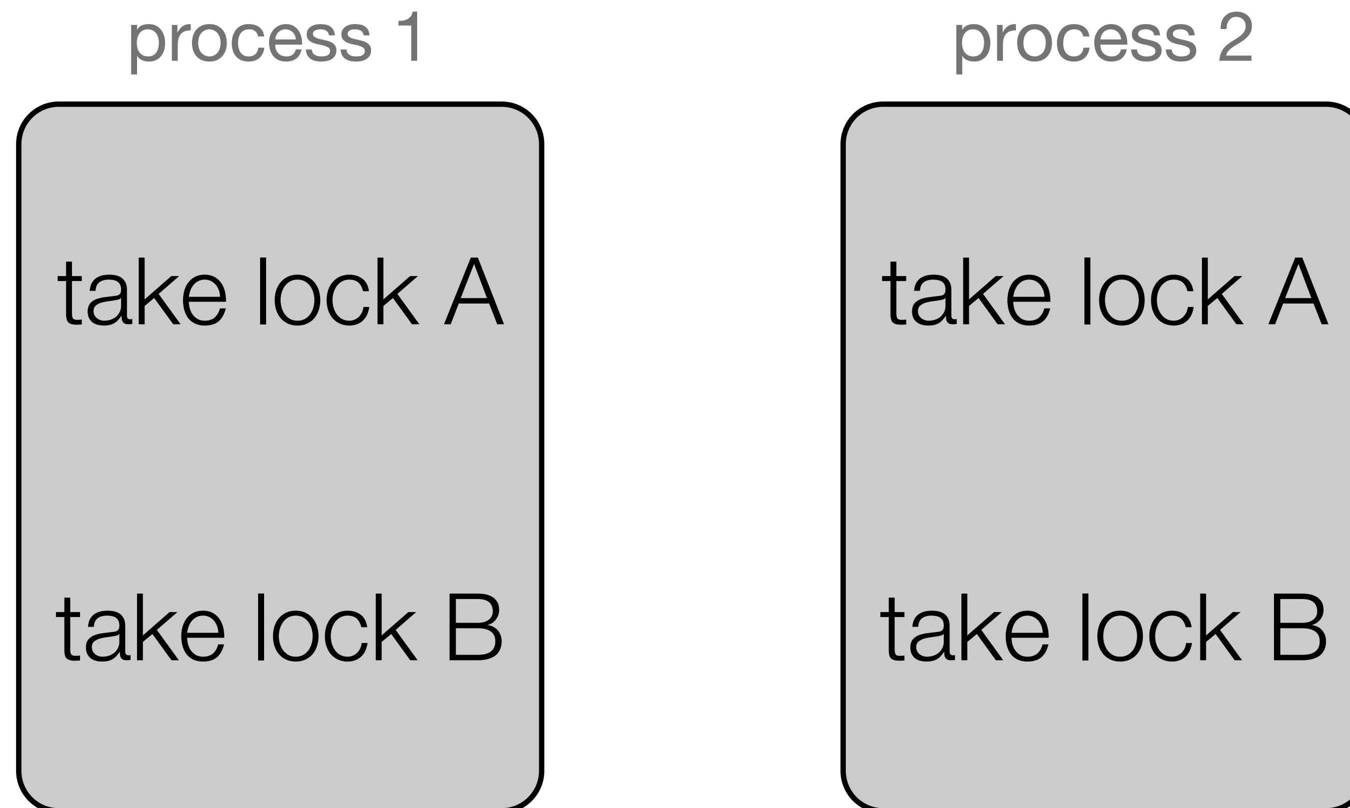
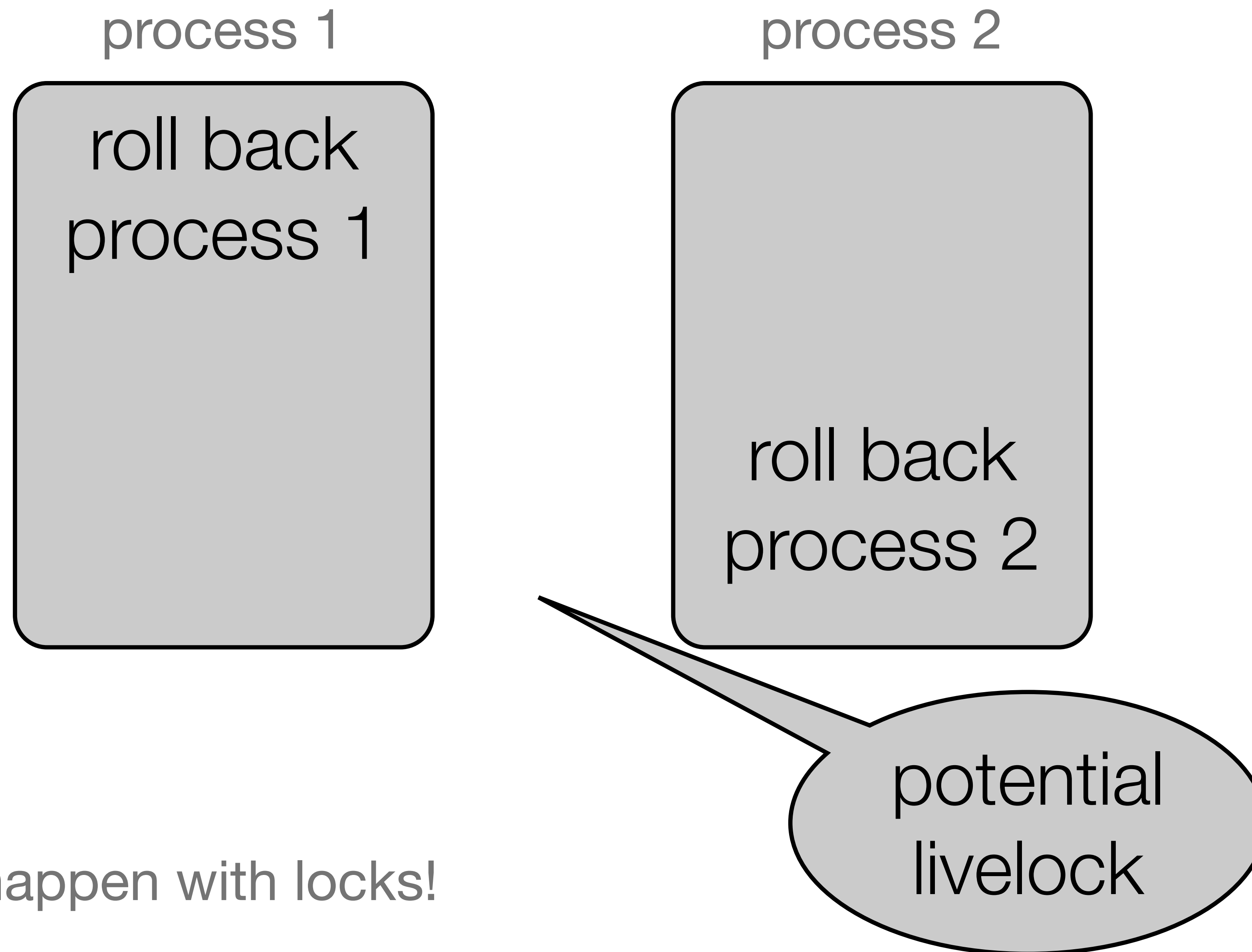
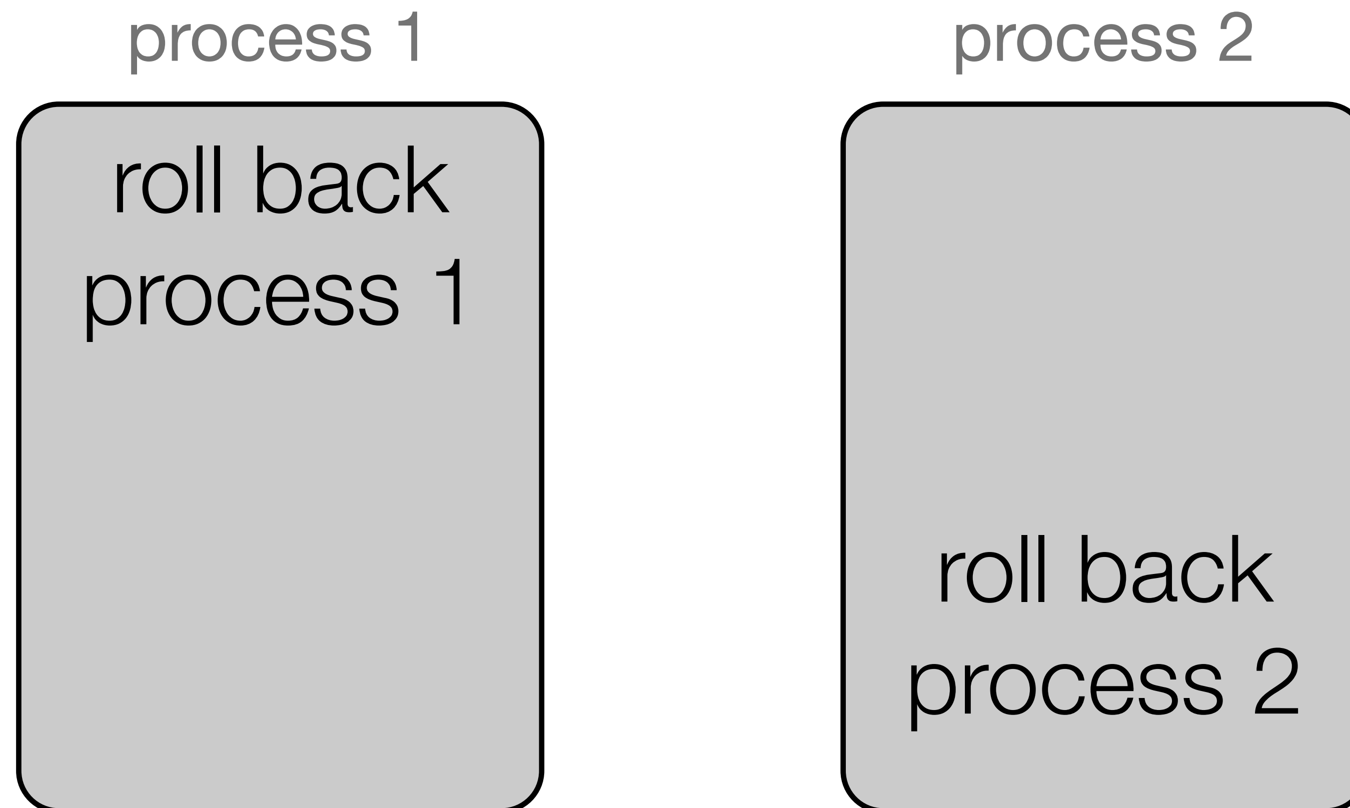
# Why is parallel programming hard?

```
(defun count-3s (list)
  (let ((count 0)) (iteration-count 0)
        (lock (mp:make-lock)))
    (dolist (number list)
      (let ((number number))
        (prun (lambda ()
                (mp:with-lock (lock)
                  (when (= number 3) (incf count))
                  (incf iteration-count))))))
    (loop until (= iteration-count (length list)))
    count)
```

Why is parallel programming hard?

- ```
  (defun count-3s (list)
    (let ((count 0)) (iteration-count 0)
          (lock (mp:make-lock)))
      (dolist (number list)
        (let ((number number))
          (prun (lambda ()
                  (mp:with-lock (lock)
                    (when (= number 3) (incf count))
                    (incf iteration-count))))))
      (loop until (= iteration-count (length list)))
      count)
  ```

- Result is now correct, but this is horribly inefficient code.

- Particularly, there is no parallelism here.
  ```

What does that mean?

- Sequential programs are much easier to understand.
- Primary goal of parallel programming: efficiency!!!
- If you don't need efficiency, don't go parallel.

Concurrency



- Important abstractions:
  Actor model, event loops, threads, locks, transactions, …

Concurrency



- Concurrent applications solves problems that are inherent in their domain.
  This is independent from the availability of parallel hardware!
  Single-core processors would have to solve these issues as well!

Parallelism



- Important abstractions:
  Parallel loops, data parallelism, fork/join, mapreduce, …

Parallelism



- Parallel programming helps improving performance.
  This is independent from the presence of concurrency in the applications!
  Single-core processors would just execute a sequential program!

Concurrency vs. parallelism

- Solutions that are good for concurrent problems may be detrimental for parallelism and vice versa.

Problems with locking...



- This cannot happen with STM!

Problems with locking...



- Solution: Take all locks always in the same order.
  (May be incorrect depending on problem domain,
  but is usually correct when only performance matters.)

Problems with STM...



- This cannot happen with locks!

Problems with STM...



- Solution: Add heuristics and back-off strategies.
  (This hurts performance even more,
  so may not be acceptable if you use parallelism for performance!)

Example: Task parallelism

- ```
  (defun count-3s (vector)
    (labels ((recur (start end)
               (let ((length (- end start)))
                 (case length
                   (0 0)
                   (1 (if (= (svref vector start) 3) 1 0))
                   (t  (let* ((half (ash length -1))
                              (middle (+ start half))))
                         (let (left right)
                           (spawn (left) (recur start middle))
                           (setq right (recur middle end))
                           (sync)
                           (+ left right)))))))
      (recur 0 (length vector))))
  ```
  ```

Parallelism for performance: Amdahl's Law

- n: number of threads
  P: parallel fraction of the algorithm
  (T n): time the algorithm takes on n threads
  (S n): speedup on n threads

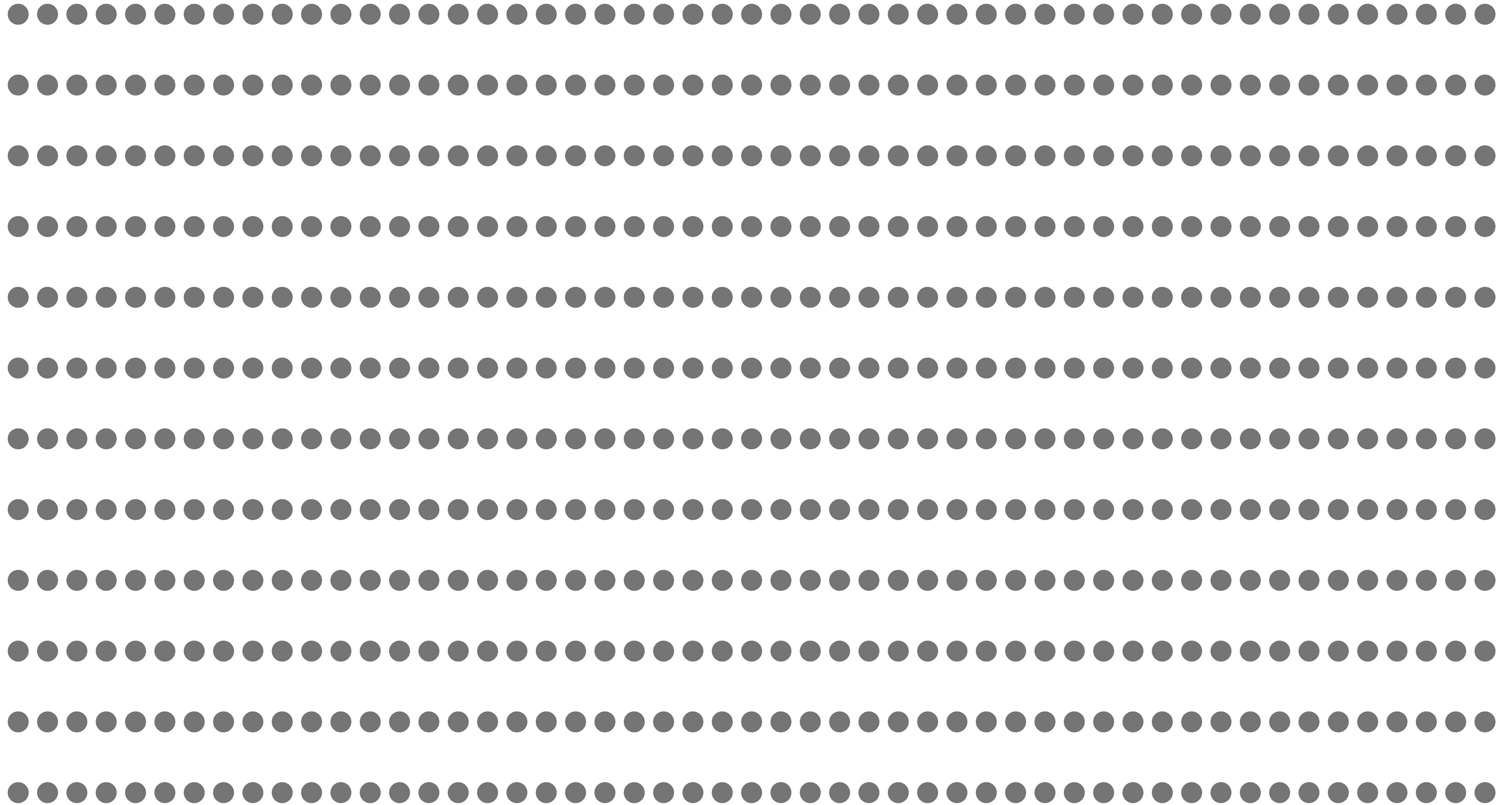
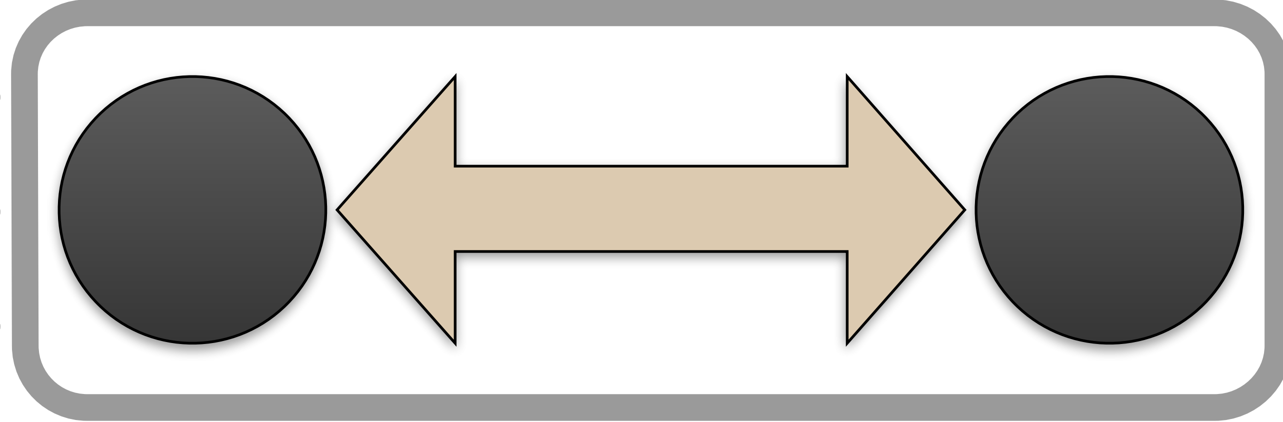
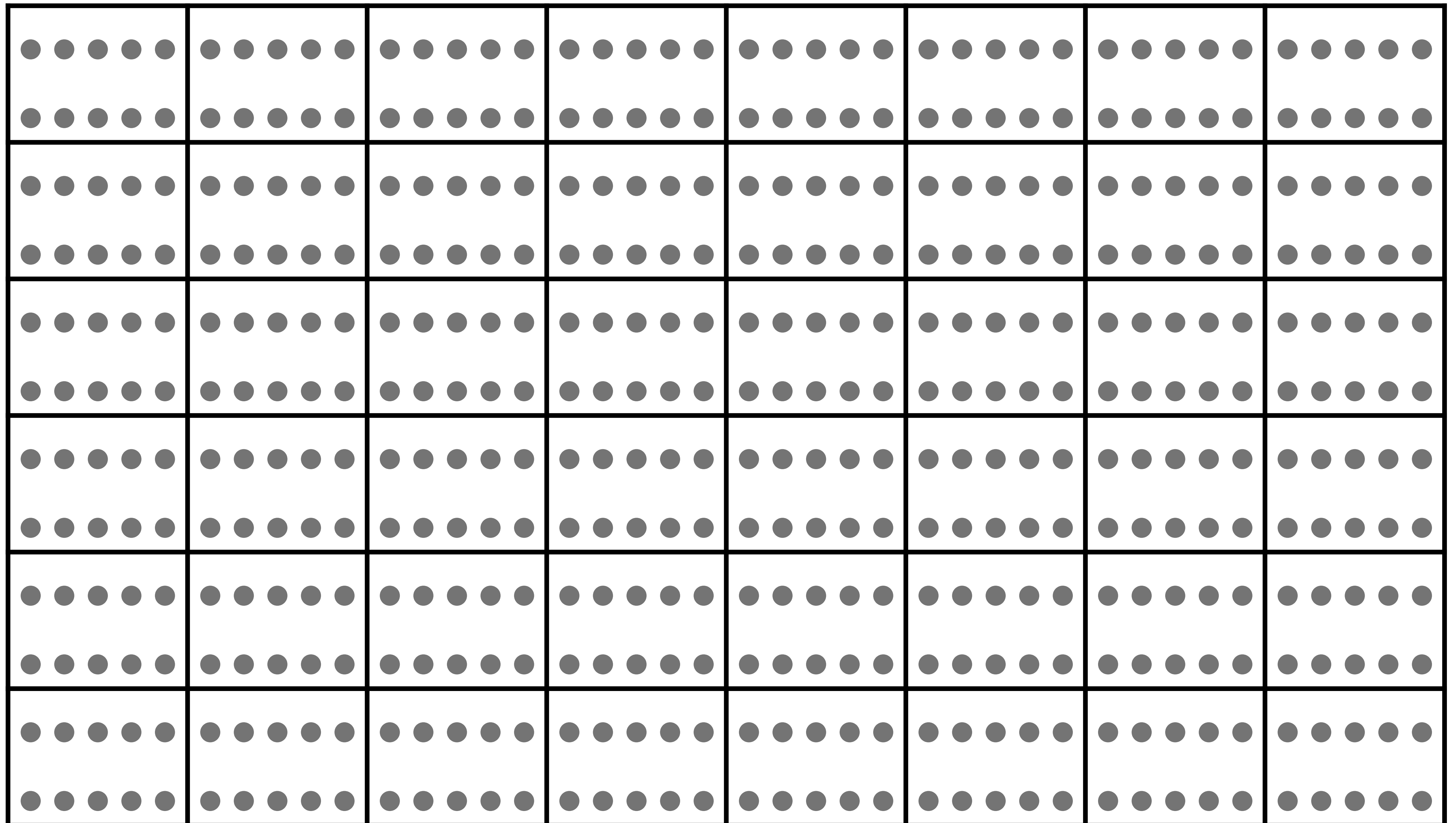
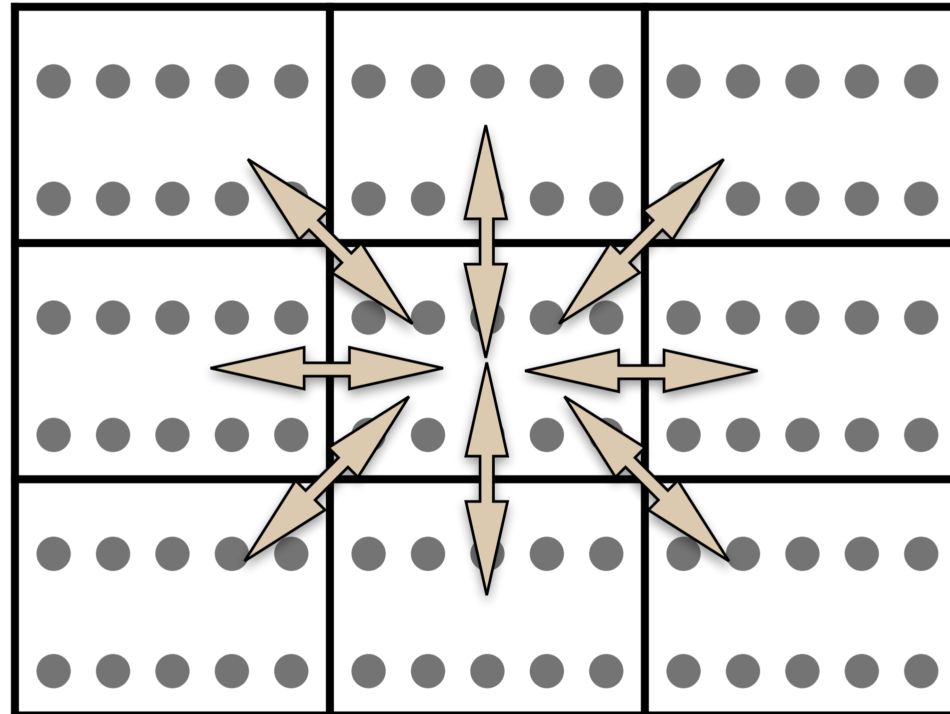
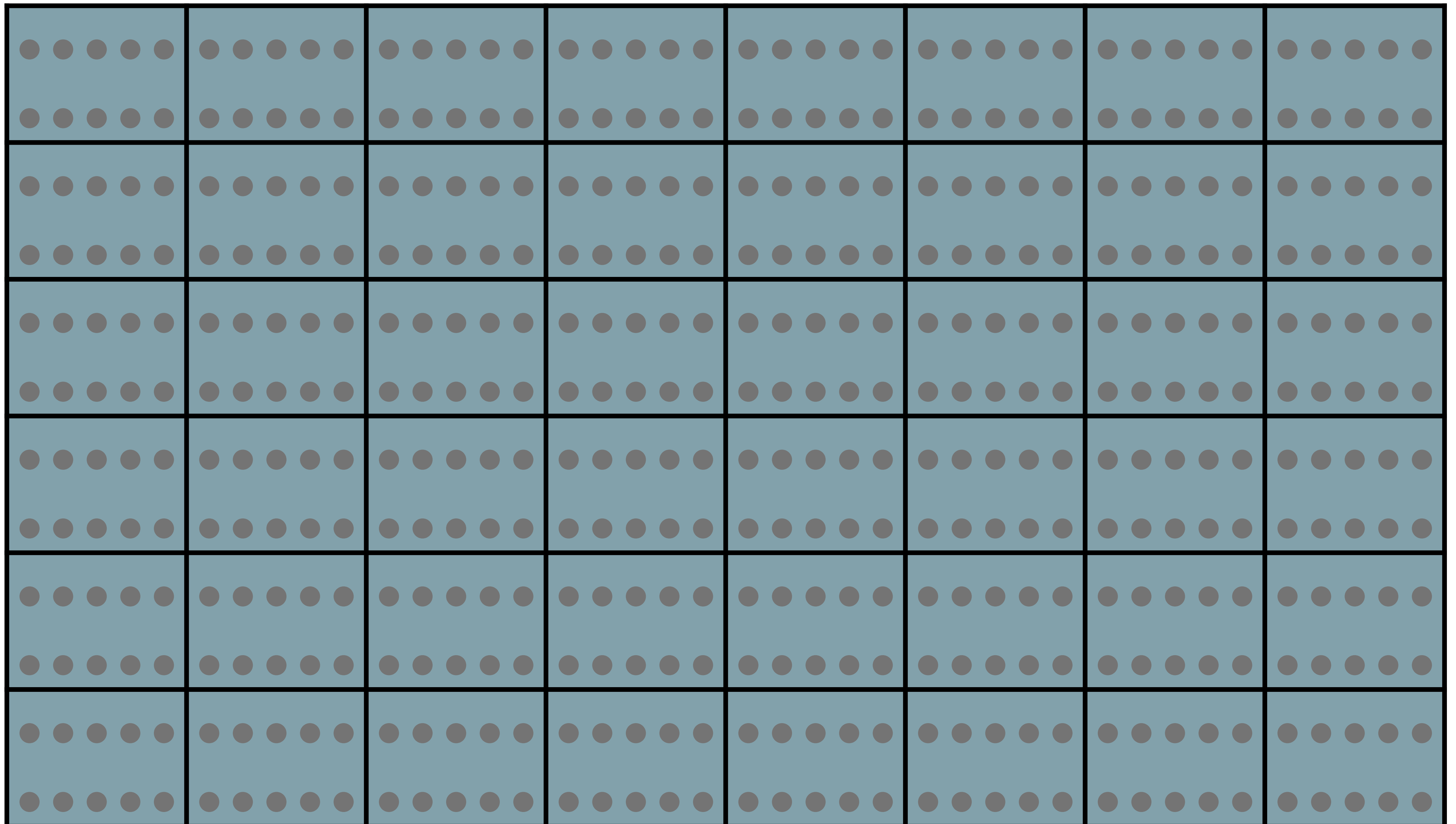
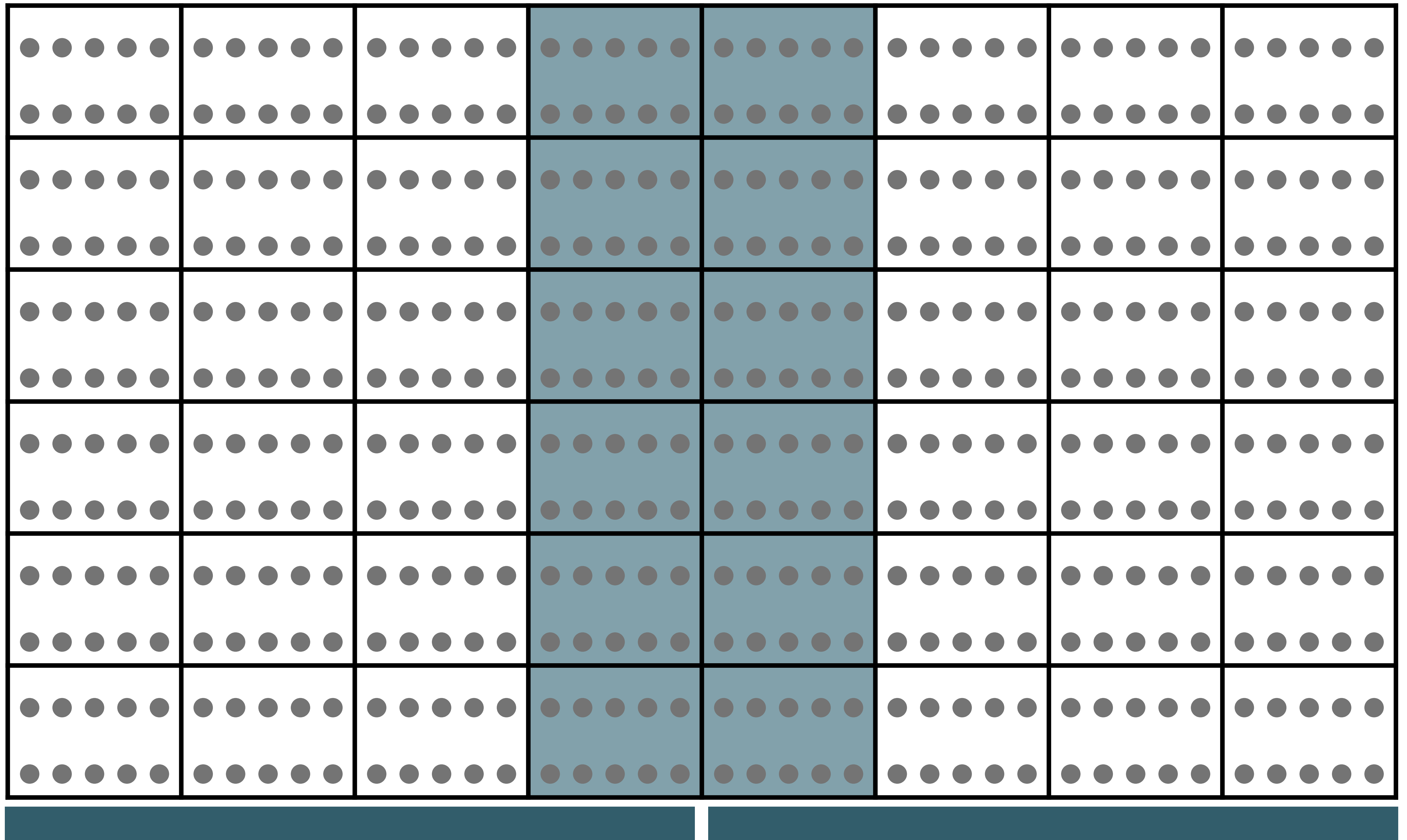
  ```
  (T n) = (let ((seqT (* (T 1) (- 1 P)))
                (parT  (* (T 1) P)))
            (+ seqT (/ parT n)))

  (S n) = (/ 1 (+ (- 1 P) (/ P n)))

  (S ∞) = (/ 1 (- 1 P))
  ```

- Examples: P = 90% => maximum speedup = 10
  P = 80% => maximum speedup = 5

Parallelism for performance: Gustafson's Law

- a: sequential time
  b: parallel time
  P: number of processors

  ```
  T(P) = (+ a (* P b))
  S(P) = (/ (+ a (* P b)) (+ a b))
  ```

- ```
  α = (/ a (+ a b))
  S(P) = (+ α (* P (- 1 α)))
       = (- P (* α (- P 1)))
  ```
  ```

Parallelism for performance

- Sequential performance is important!
  The better the sequential section performs,
  the better the gains from parallelization!

General performance considerations in Lisp

- Avoid consing / memory management
  => dynamic extent / stack allocation
  => boxed vs. unboxed types
  => 64bit vs. 32bit Lisp implementations
  => control the garbage collector

- Identify and optimize fast paths

- Inline declarations

- Optimization & type declarations

- typed aref, int32 operations (LispWorks-specific)

Recipe for parallelization

- Identify the parallelism
- Identify the dependencies
  - Identifying parallelism is relatively easy.
    Identifying dependencies is harder.
    Handling dependencies efficiently is even harder.
    => Know your toolbox!

Synchronization toolbox in LispWorks

- Mailboxes
- Locks, incl. shared locks
- Hash tables, vectors with fill pointers
- Barriers, semaphores, condition variables,
  atomic operations, memory order synchronization
- Thread-local memory allocation
- Garbage collector freezes all processes

Example: fluidanimate

- One of the benchmarks
  in the PARSEC benchmark suite

- Implementation of the
  Smoothed-particle Hydrodynamics method

Example: fluidanimate



Example: fluidanimate



Example: fluidanimate



Example: fluidanimate



Example: fluidanimate



Example: fluidanimate



Fluidanimate: data structures

- ```
  (defstruct (vec3 (:constructor make-vec3 (x y z))
                   (:constructor make-zero-vec3 ())
     (x 0.0 :type float-type)
     (y 0.0 :type float-type)
     (z 0.0 :type float-type))
  ```

- ```
  (defstruct (cell (:constructor make-cell ())
     (p       (create-array 16 ‘make-zero-vec3 :element-type ‘vec3)
              :type (array vec3 (16))))
     (hv      (create-array 16 ‘make-zero-vec3 :element-type ‘vec3)
              :type (array vec3 (16))))
     (v       (create-array 16 ‘make-zero-vec3 :element-type ‘vec3)
              :type (array vec3 (16))))
     (a       (create-array 16 ‘make-zero-vec3)
              :type simple-vector))
     (density (make-array 16 :initial-element 0.0)
              :type simple-vector))
  ```

Fluidanimate: main loop

```
• (defun advance-frame ()
    (declare #.*optimization*)
    (let ((num-grids (num-grids)))
      (declare (fixnum num-grids))
      (parallel-for num-grids ‘clear-particles-mt)
      (parallel-for num-grids ‘rebuild-grid-mt)
      (parallel-for num-grids
                         ‘init-densities-and-forces-mt)
      (parallel-for num-grids ‘compute-densities-mt)
      (parallel-for num-grids ‘compute-densities-2-mt)
      (parallel-for num-grids ‘compute-forces-mt)
      (parallel-for num-grids ‘process-collisions-mt)
      (parallel-for num-grids ‘advance-particles-mt)))
```

Fluidanimate: example step

- ```
  (defun compute-forces-mt (i)
    (declare #.*optimization*)
    (loop over each cell in (aref grids i) do
      (loop over each p in cell do
        (loop over each neighbor of cell do
          (loop over each np in neighbor do
            (when (< (index np) (index p))
              (let ((acc (compute-acc p np)))
                (if (borderp cell)
                  (loop
                     for old = (svref (cell-a cell) p)
                     for new = (vec3+ old acc)
                     until (compare-and-swap (svref (cell-a cell) p) old new))
                  (vec3+= (svref (cell-a cell) p) acc))
                (if (borderp neighbor)
                  (loop
                     for old = (svref (cell-a neighbor) np)
                     for new = (vec3- old acc)
                     until (compare-and-swap (svref (cell-a cell) np) old new))
                  (vec3-= (svref (cell-a cell) np) acc)))))))))
  ```
  ```

Fluidanimate: main challenges

- When translating from C++ to Lisp,
  we encountered the following performance challenges:
  - Too much consing.
  - Controlling the garbage collector.
  - Getting synchronization right.
  - Abstracting away the parallelization.

Fluidanimate: Too much consing.

- At the heart of the algorithm,
  lots of uses of vec3 operations:

- ```
  (defun vec3+ (v1 v2)
    (make-vec3 (+ (vec3-x v1) (vec3-x v2))
               (+ (vec3-y v1) (vec3-y v2))
               (+ (vec3-z v1) (vec3-z v2))))
  ```

- Same for subtraction, negation, multiplication, division.

- No matter the type and optimization declarations,
  there are just far too many intermediate results on the heap.
  ```

Fluidanimate: Too much consing - solution.

```
• (defstruct (vec3
               (:constructor make-vec3 (x y z))
               (:constructor make-zero-vec3 ()
               (:constructor vec3+ (v1 v2 &aux
                 (x (+ (vec3-x v1) (vec3-x v2)))
                 (y (+ (vec3-y v1) (vec3-y v2))
                 (z (+ (vec3-z v1) (vec3-z v2)))))
               (:constructor vec3* (v1 s &aux
                 . . .)
               . . .)
    (x 0.0 :type float-type)
    (y 0.0 :type float-type)
    (z 0.0 :type float-type))
```

Fluidanimate: Too much consing - solution.

- Now the return values can be declared
  with dynamic-extent at the receiving side,
  and will therefore be allocated on the stack,
  not on the heap.

- However, intermediate results must all be named.

- Inline functions with calls to such constructors *may* also work with dynamic-extent declarations and result stack allocation.

Fluidanimate:
Stack allocation for intermediate results would be nice.

```
• ...
  (vec3*= acc pressure-coeff)
  (vec3*= acc (/ (* hmr hmr) dist))
  (vec3*= acc (- (+ (svref cell-density cell) j))
                    (svref cell-density neigh) i)))
                 double-rest-density))
  (let* ((cell-v (aref (cell-v cell) j))
         (neigh-v (aref (cell-v neigh) i))
         (ddd (vec3- neigh-v cell-v)))
    (declare (vec3 cell-v neigh-v ddd)
             (dynamic-extent ddd))
    (vec3*= ddd viscosity-coeff)
    (vec3*= ddd hmr)
    (vec3+= acc ddd)
    (vec3/= acc (* (svref (cell-density cell) j)
                   (svref (cell-density neigh) i)))
    ...
```

return-with-dynamic-extent

- My wish:

- ```
  (defun vec3+ (v1 v2)
    (return-with-dynamic-extent
      (make-vec3 (+ (vec3-x v1) (vec3-x v2))
                 (+ (vec3-y v1) (vec3-y v2))
                 (+ (vec3-z v1) (vec3-z v2)))))
  ```

- [Also: dynamic extent for typed aref vectors!]
  ```

Fluidanimate: Abstracting away parallelization.

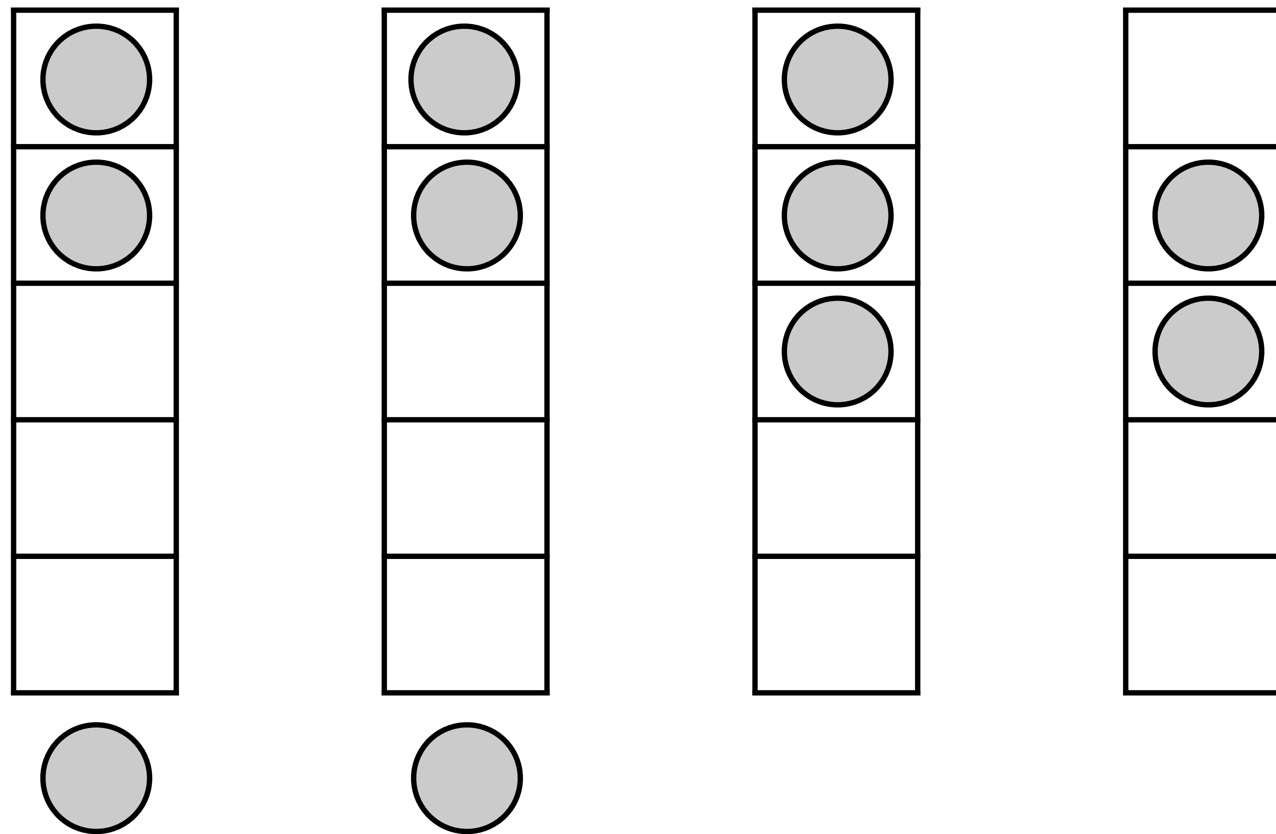
- (defun parallel-for (numgrids function)
```
   (loop for i below numgrids
         collect (mp:process-run-function
                    (string function) ‘() function i)
         into processes finally
         (mapc ‘mp:process-join processes)))
```

- OK when numgrids = available processors.

- Otherwise distribute grids evenly
  over available processor. (Easy.)

- However, what if there is load imbalance?
  => Employ work stealing!

What is work stealing?



- Each thread has a deque, pushes and pops work on one end;
  steals work from other queues on other ends;
  stealing occurs only when own deque is empty
  => stealing is a rare event, work is mostly done on local data.

Work stealing with fork/join computations. (Inspired by Cilk, here with claws.)

- Primitives:
  - spawn = add work to local deque
  - sync = ensure all previously local work is done
  - reset-workers = create n worker threads
  - all stealing happens in the background

parallel-for with fork/join

- ```
  (defun parallel-for (n function)
    (labels ((recur (k start end)
               (let ((diff (- end start)))
                 (case diff
                   (0)
                   (1 (funcall function start))
                   (t (let* ((half (ash diff -1))
                             (middle (+ start half)))
                        (spawn () (recur middle end))
                        (recur start middle)
                        (sync)))))))
      (recur 0 n))
  ```
- aka divide and conquer
  ```

Example: sequencing pipelines

- DNA sequencing ≈ string pattern matching
- Sequencing pipelines:
    1. Match reads against a reference
    2. Analyse matched reads

DNA ≈ Code

- DNA: AGGCTACTTAAAT… => genome

- RNA: GUGCAUUGAGUA… => transcriptome

- Protein: V H L T P E E E K…

DNA Sequencing ≈ Pattern matching

+

≠ A G G C T A C T T A A A T…

= A G G C

C T A

G T T

A A T…

↔ A G C C T A A T T G A A T…

Alignment ≈
Where do the reads
match in the reference?

SAM/BAM Files

- { QNAME: “ERR091575.2”,
  FLAG: 16,
  RNAME: “chr13”,
  POS: 100163268
  MAPQ: 37,
  CIGAR: 100M,
  RNEXT: “*”,
  PNEXT: 0,
  TLEN: 0,
  SEQ: “GCTTCTCCTGAGATCATCG…”,
  QUAL: “DDDDDDDDDDEDEDDDDD…”,
  XT: “U”, NM: 3, X0: 1, X1: 0, XM: 3, XO: 0, XG:0,
  MD: “25C17A55T0”, RG: “group1” }

CIGAR Strings

Op	BAM	Description
M	0	alignment match (can be a sequence match or mismatch)
I	1	insertion to the reference
D	2	deletion from the reference
N	3	skipped region from the reference
S	4	soft clipping (clipped sequences present in SEQ)
H	5	hard clipping (clipped sequences NOT present in SEQ)
P	6	padding (silent deletion from padded reference)
=	7	sequence match
X	8	sequence mismatch

- 10H30M15I32M8H

Typical computation on CIGAR Strings

- ```
  (defun sum-referenced (cigar)
    (loop for (key . val) in cigar
          if (member key ‘(:M :D :N := :X))
          sum val fixnum))
  ```
  ```

Better version

- ```
  (define-symbol-macro cigar-ops "MIDNSHPX=")
  (defconstant +min-cigar-op+
    (reduce 'min cigar-ops :key 'char-code))
  (defconstant +max-cigar-op+
    (reduce 'max cigar-ops :key 'char-code))
  ```

- ```
  (declaim (inline cigar-index))

  (defun cigar-index (char)
    (declare (base-char char)
             (optimize (speed 3) (space 0) (debug 0) (safety 0)
                       (compilation-speed 0) (fixnum-safety 0)))
    (- (char-code char) +min-cigar-op+))
  ```

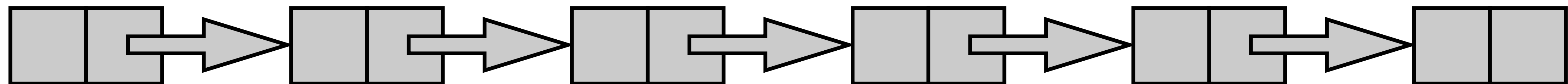
Better version

- ```
  (defun make-reference-table ()
    (let ((table (make-array (1+ (- +max-cigar-op+
                                      +min-cigar-op+))
                  :initial-element #.(code-char 0)
                  :element-type ‘base-char
                  :allocation :long-lived
                  :single-thread t)))
     (flet ((set-reference (char)
              (setf (sbchar table (cigar-index char)) #.(code-char 1))))
       (set-reference #\M)
       (set-reference #\D)
       (set-reference #\N)
       (set-reference #\=)
       (set-reference #\X))))
  ```
  ```

Better version

- (declaim (notinline sum-referenced))

```
(defun sum-referenced (cigar)
  (declare (list cigar)
     (optimize (speed 3) (space 0) (debug 0) (safety 0)
                    (compilation-speed 0) (fixnum-safety 0)))
  (let ((table (load-time-value (make-reference-table) t)))
   (loop for (key . value) in cigar
          for keychar = (sbchar (symbol-name key) 0)
          for factor = (char-code (sbchar table keychar))
          sum (* factor value) fixnum)))
```

Summary

- Distinguish concurrency vs. parallelism
- If you want parallelism for performance,
  you need to care about sequential performance as well.
- There is no single parallel programming paradigm
  that fits all problems.
- Parallel Lisps should provide building blocks
  for domain-specific parallel constructs
  (or just constructs suited for different domains).

What next?

- Work stealing (Intel Cilk Plus, TBB)
- Intel Cilk Plus: CEAN (Paralation Lisp, APL)
- Intel CnC (incl. distributed processing)
- MPI
- PGAS: Chapel, Fortress, X10

Thank You

Why are Lisp programs hard to parallelize?

- ```
  (defun count-3s (list)
    (if (null list)
      0
      (+ (if (= (car list) 3) 1 0)
         (count-3s (cdr list)))))
  ```
  ```

Why are Lisp programs hard to parallelize?

- ```
  (defun count-3s (list)
    (if (null list)
      0
      (+ (if (= (car list) 3) 1 0)
         (count-3s (cdr list)))))
  ```

- Lists from cons cells are an inherently sequential data structure.


  ```

Mailboxes out of condition variables

```
(defstruct mailbox
  (cond (mp:make-condition-variable))
  (lock (mp:make-lock))
  (list '()))

(defun mailbox-read (mailbox)
  (mp:with-lock ((mailbox-lock mailbox))
    (loop until (mailbox-list mailbox) do
        (mp:condition-variable-wait
            (mailbox-cond mailbox) (mailbox-lock mailbox)))
    (pop (mailbox-list mailbox))))

(defun mailbox-send (mailbox object)
  (mp:with-lock ((mailbox-lock mailbox))
    (push object (mailbox-list mailbox))
    (mp:condition-variable-signal (mailbox-cond mailbox))))
```

Barriers

- ```
  (defun count-3s (vec &aux (length (length vec)))
    (loop with block-size = (ceiling length *threads*)
          with mailbox = (mp:make-mailbox)
          with barrier = (mp:make-barrier *threads*)

          for start below length by block-size
          for end = (min (+ start block-size) length) do

          (prun (lambda (start end)
                  (loop for i from start below end do
                        (setf (svref vec i) (if (= (svref vec i) 3) 1 0)))
                  (mp:barrier-wait barrier)
                  (mp:mailbox-send mailbox
                    (loop for i from start below end sum (svref vec i)))))

          finally (return (loop repeat *threads* sum (mp:mailbox-read mailbox)))))
  ```
  ```

Fork/join + work stealing

- Fork/join and work stealing are independent of each other.
  (For example, work stealing schedulers also useful for futures, actors, ...)

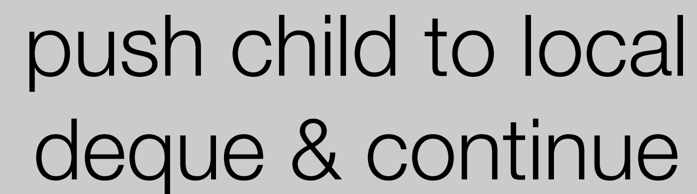
- Literature shows optimal theoretical performance and space bounds when fork/join and work stealing are combined.
  (Blumofe, Leiserson,Scheduling Multithreaded Computations by Work Stealing, FOCS'94 & J. ACM'99)

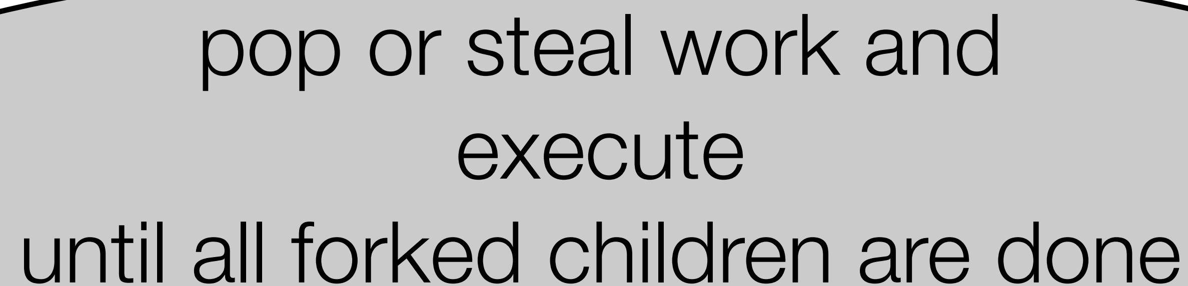
- Confirmed by practical experience (Cilk, TBB, Java fork/join, ...)
  (Frigo, Leiserson, Randall, The Implementation of the Cilk-5 Multithreaded Language, PLDI'98)

- Work stealing works well until a certain number of processor cores.
  (Dinan et al., Scalable Work Stealing, SC'09)

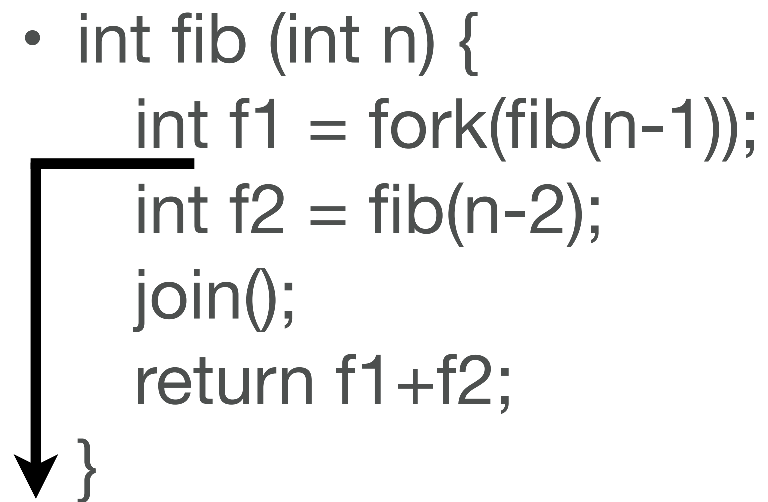
Library-style work stealing (TBB, Java fork/join, ...)

- ```
  int fib (int n) {
    int f1 = fork(fib(n-1));
    int f2 = fork(fib(n-2));
    join();
    return f1+f2;
  }
  ```



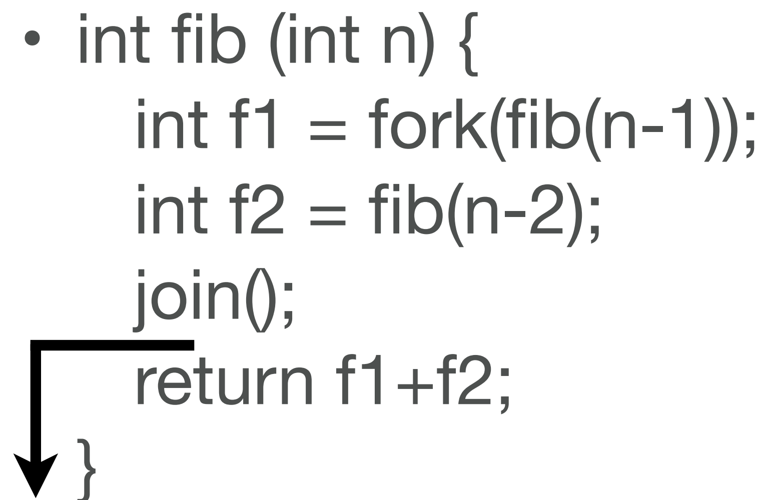

  ```

Cilk-style work stealing

- ```
  int fib (int n) {
     int f1 = fork(fib(n-1));
     int f2 = fork(fib(n-2));
     join();
     return f1+f2;
  }
  ```

push continuation to local deque;
fork-count++, execute child, fork-count--;
if child finishes a join, execute its continuation

if all children done, just continue;
otherwise forget about the join,
instead steal and execute work.
  ```

The continuation of a fork...



```
int fib (int n) {
    int f1 = fork(fib(n-1));
    int f2 = fib(n-2);
    join();
    return f1+f2;
}
```

...and the continuation of a join.



- int fib (int n) {
    int f1 = fork(fib(n-1));
    int f2 = fib(n-2);
    join();
    return f1+f2;
  }

Cilk-style vs. library-style work stealing

- Cilk-style guarantees optimal performance *and* optimal space bounds
  - ...requires efficient representation of continuations & compiler support
- Library-style guarantees only optimal performance, no space bounds
  - ...requires trickery in join step to support better space bounds (TBB)
- *(Guo, Barik, Raman, Sarkar, Work-first and help-first scheduling policies for async-finish task parallelism, IPDPS'09)*