TDDD56: Multicore and GPU programming
Lesson 1
Introduction to laboratory work

Nicolas Melot
nicolas.melot (at) liu.se

Linköping University
Dept. of Computer and Inf. Science
Linköping, Sweden

October 27, 2016

Today

Lab 1: Load balancing

- Parallelize the generation of the graphic representation of a subset of Mandelbrot set.

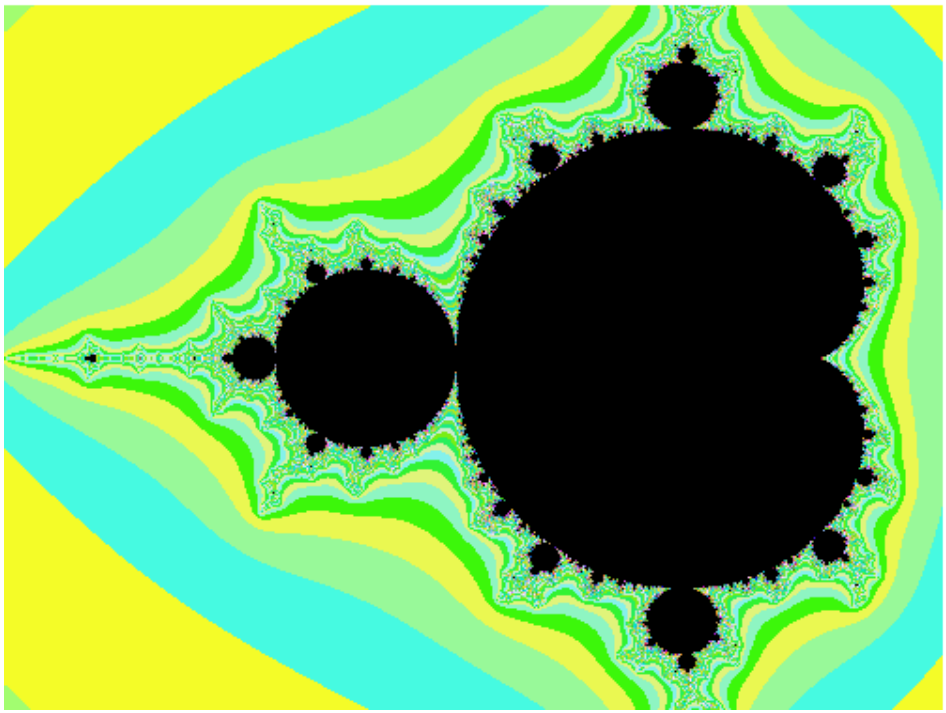


Figure: A representation of the Mandelbrot set (black area) in the range $-2 \leq C^{re} \leq 0.6$ and $-1 \leq C^{im} \leq 1$.

Computing 2D representation of the Mandelbrot set

- Let $P = \{(a, b \cdot i) : a \in [C^{re}_{min}, C^{re}_{max}] \wedge b \in [C^{im}_{min}, C^{im}_{max}]\} \subset \mathbb{C}$.
- Represent P in a 2D picture of size $H \times W$ pixels.
- For each $p \in P$, we have exactly one pixel $(x, y) = (\frac{a \cdot H}{C^{re}_{max} - C^{re}_{min}}, \frac{b \cdot W}{C^{im}_{max} - C^{im}_{min}})$ in the picture
- $p \in M$ iif the Julia sequence (eq. 1) is bound to $b \in \mathbb{R}$ with $b > 0$ for $c = p$ starting at $z = (0, 0i)$.

$$\begin{cases} a_0 & = z \\ a_{n+1} & = a_n^2 + c, \forall n \in \{\forall x \in \mathbb{N} : n \neq 0\} \end{cases} \quad (1)$$

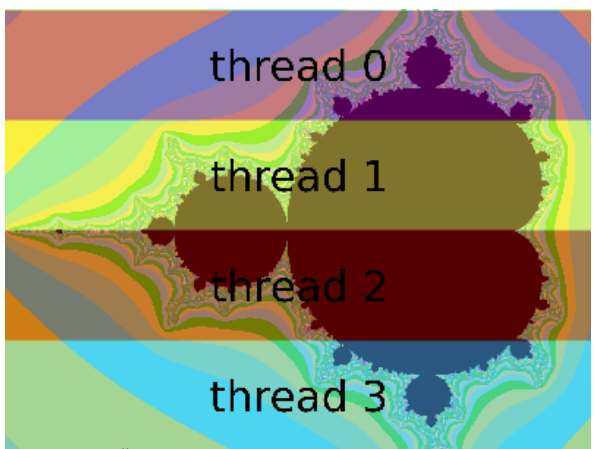
- Iterative algorithm implemented and provided

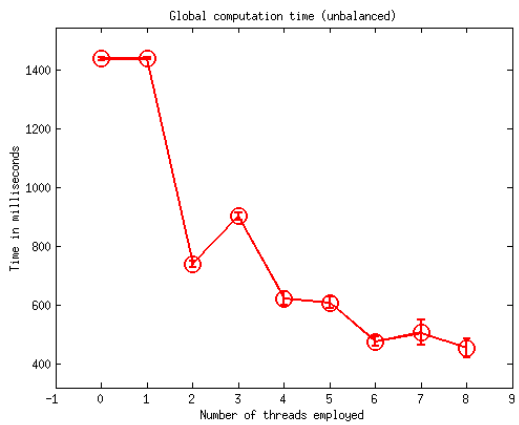
Computing 2D representation of the Mandelbrot set

```
int is_in_Mandelbrot(float Cre, float Cim) {
    int iter;
    float x=0.0, y=0.0, xto2=0.0, yto2=0.0, dist2;
    for (iter = 0; iter <= MAXITER; iter++) {
        y = x * y;
        y = y + y + Cim;
        x = xto2 – yto2 + Cre;
        xto2 = x * x;
        yto2 = y * y;
        dist2 = xto2 + yto2;
        if ((int) dist2 >= MAXDIV) {
            break; // diverges
        } }
    return iter;
}
```

Computing 2D representation of the Mandelbrot set

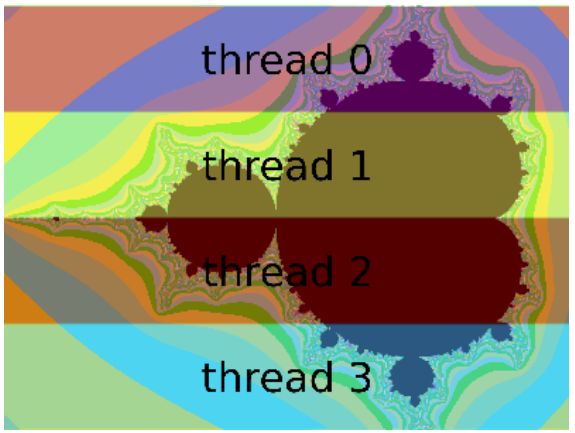
- *is_in_mandelbrot*(C^{re}, iC^{im}) returns n
  - If $n >= MAXITER$ then the pixel is black
  - Otherwise the pixel takes the n^{th} color of a *MAXITER* color gradient.
- Take each pixel (h, w), run *is_in_mandelbrot*(h, w) and deduce a suitable color
- Data-parallel, “embarassingly parallel”
- Load-balancing issues

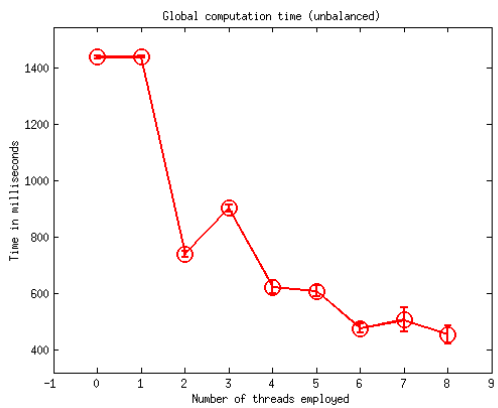




Lab 1: Load balancing

- A sequential code is provided
- Parallelize it using threads and pthread library.
  - Naive: partition work as upper right figure. Performance like lower right figure
  - Load-balanced: performance scale with number of threads.
    - Independant from *P* to compute
  - Measure individual threads' execution time. Compare global naive and balanced execution time.
- Hints in necessary modifications:
  - *mandelbrot.c*: modify from line 122 to 146
  - You may modify anything else if you want
- More details and hints in lab compendium (read it!)





Lab 2: Non blocking data structures

- Synchronization of stacks
- Stack: LIFO (Last In, First Out) data structure with *push* and *pop* operations
  - Often a single linked list
  - *head* points to the head element
  - Empty if *head* $\rightarrow$ *null*

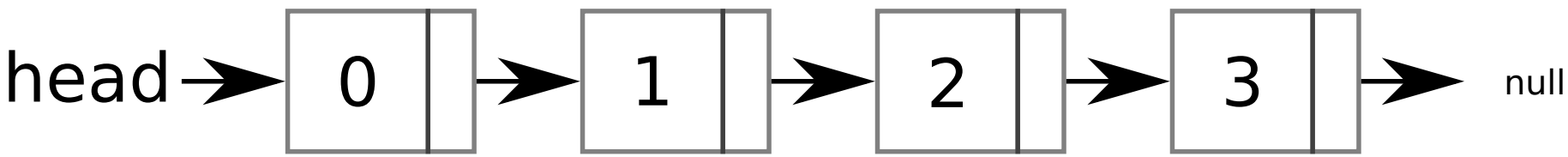


Figure: Unbounded stack

Stack synchronization

- Protects the stack during *push* or *pop* operations
- Atomically update head pointer
- Protection usually achieved thanks to a lock.

```
P(lock);
new_element.next = head;
head = &new_element;
V(lock);
```

Figure: Stack lock-based synchronization

Compare-and-swap

- Do atomically:
  - If *pointer* $\neq$ *old pointer* then return false
  - Else swap *pointer* to *new pointer* and return true

```
CAS(void** buf, void* old, void* new) {
   atomic {
      if(*buf = old) {
         *buf = new;
      }
   }
   return old;
}
```

- How to protect a stack with it?

CAS-protected stack

- Keep track to old head
- Set new element's *next* member to current head saved to old
- Check if head is still the one recorded in old
  - If so, commit the change
  - Else, restart the process (keep track, update new, check) until commit

```
do {
   old = head;
   elem.next = old;
} while(CAS(head, old, elem) != old);
```

Figure: CAS-protected push

CAS-protected stack

- If another thread preempts and push an element
  - Head is changed
  - CAS will fail
  - First thread tries again
- Thread 1 gets preempted by thread 2

```
   do {
      old = head;
      elem1.next = old;
<thread 2 preempts thread 1>
   } while(CAS(head, old, elem1) != old);

<just preempted thread 1>
   {
      old = head;
      elem2.next = old;
   } while(CAS(head, old, elem2) != old);
<returns to thread 1>
```

Misuses case of CAS

Why is the code below wrong...

```
push(stack_t stack, elem_t elem) {
   do {
      elem.next = head;
      old = head;
   } while(CAS(head, old, elem) != old);
}
```

... while this one is correct?

```
   do {
      old = head;
      elem.next = old;
   } while(CAS(head, old, elem) != old);
```

Treiber stack

- Rather old: published by R. Kent Treiber from IBM research in 1986.
  - Search “Coping with parallelism” on http://domino.research.ibm.com/library/cyberdig.nsf/index.html
- Relies on hardware CAS to atomically update the head of a stack to a new element.
- Pseudo code available on page 6 of “Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors” (M.M. Michael, M.L. Scott, *Journal of parallel and distributed computing*, 1998), available through google scholar.

The ABA problem

- Consider an unbounded stack
- Consider the element *pop*'ed elements are not destroyed, but pushed in a pool to be reused in further *push*()
  - Save a *malloc* to allocate a new element, when push and pop are frequents.
- Shared memory programming
- 3 threads, each having their own pool of unused elements.

The ABA problem

- The scenario starts
  - Thread 0
    - old→null
    - new→null
    - pool→null
  - Thread 1
    - old→null
    - new→null
    - pool→null
  - Thread 2
    - old→null
    - new→null
    - pool→null
- shared→A→B→C→null

The ABA problem

- The scenario starts
- Thread 0 pops A, preempted before CAS(head, old=A, new=B)
- shared→A→B→C→null

- Thread 0
  - old→A
  - new→B
  - pool→null
- Thread 1
  - old→null
  - new→null
  - pool→null
- Thread 2
  - old→null
  - new→null
  - pool→null

The ABA problem

- The scenario starts
- Thread 0 pops A, preempted before CAS(head, old=A, new=B)
- Thread 1 pops A, succeeds

- Thread 0
  - old→A
  - new→B
  - pool→null
- Thread 1
  - old→A
  - new→B
  - pool→A→null
- Thread 2
  - old→null
  - new→null
  - pool→null

- shared→B→C→null

The ABA problem

- The scenario starts
- Thread 0 pops A, preempted before CAS(head, old=A, new=B)
- Thread 1 pops A, succeeds
- Thread 2 pops B, succeeds

- Thread 0
  - old→A
  - new→B
  - pool→null
- Thread 1
  - old→A
  - new→B
  - pool→A→null
- Thread 2
  - old→B
  - new→C
  - pool→B→null

- shared→C→null

The ABA problem

- The scenario starts
- Thread 0 pops A, preempted before CAS(head, old=A, new=B)
- Thread 1 pops A, succeeds
- Thread 2 pops B, succeeds
- Thread 1 pushes A, succeeds

- Thread 0
  - old→A
  - new→B
  - pool→null
- Thread 1
  - old→C
  - new→A
  - pool→null
- Thread 2
  - old→B
  - new→C
  - pool→B→null

- shared→A→C→null

The ABA problem

- The scenario starts
- Thread 0 pops A, preempted before CAS(head, old=A, new=B)
- Thread 1 pops A, succeeds
- Thread 2 pops B, succeeds
- Thread 1 pushes A, succeeds
- Thread 0 performs CAS(head, old=A, new=B)

- shared→B→null

- Thread 0
  - old→A
  - new→B
  - pool→A→null
- Thread 1
  - old→C
  - new→A
  - pool→null
- Thread 2
  - old→B
  - new→C
  - pool→B→null

The ABA problem

- The scenario starts
- Thread 0 pops A, preempted before CAS(head, old=A, new=B)
- Thread 1 pops A, succeeds
- Thread 2 pops B, succeeds
- Thread 1 pushes A, succeeds
- Thread 0 performs CAS(head, old=A, new=B)
- The shared stack should contain C element only, but head points to B in Thread 2's recycling bin
  - shared→B→null

- Thread 0
  - old→A
  - new→B
  - pool→A→null
- Thread 1
  - old→C
  - new→A
  - pool→null
- Thread 2
  - old→B
  - new→C
  - pool→B→null

The ABA problem

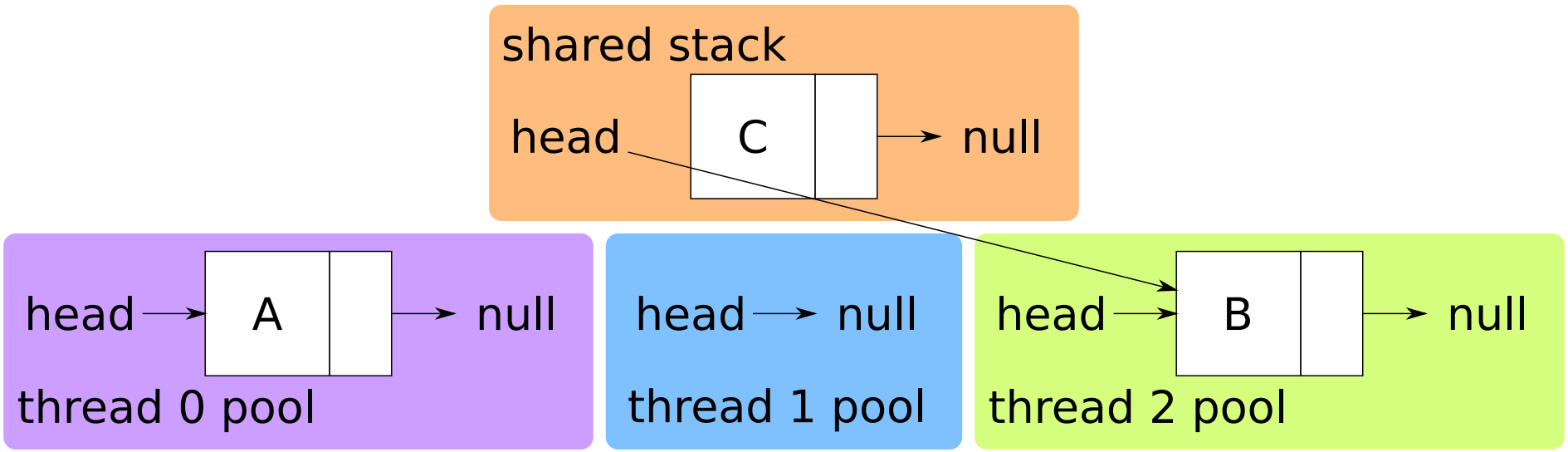


Figure: The shared stack should contain C element only, but head points to B in Thread 2's recycling bin.

Lab 2: Directions

Direction

- Implement a stack and protect it using locks

Lab 2: Directions

Direction

- Implement a stack and protect it using locks
- Implement a CAS-based stack
  - A CAS assembly implementation is provided in the lab skeleton

Lab 2: Directions

Direction

- Implement a stack and protect it using locks
- Implement a CAS-based stack
  - A CAS assembly implementation is provided in the lab skeleton
- Use pthread synchronization to make several threads to preempt each other in order to play one ABA scenario

Lab 2: Directions

Direction

- Implement a stack and protect it using locks
- Implement a CAS-based stack
  - A CAS assembly implementation is provided in the lab skeleton
- Use pthread synchronization to make several threads to preempt each other in order to play one ABA scenario
- Use a ABA-free performance test to compare performance of a lock-based and CAS-based concurrent stack

Lab 2: Directions

Direction

- Implement a stack and protect it using locks
- Implement a CAS-based stack
  - A CAS assembly implementation is provided in the lab skeleton
- Use pthread synchronization to make several threads to preempt each other in order to play one ABA scenario
- Use a ABA-free performance test to compare performance of a lock-based and CAS-based concurrent stack
  - Warning: malloc/free operations expensive: make sure they don't occur in performance test

Lab 2: Directions

Direction

- Implement a stack and protect it using locks
- Implement a CAS-based stack
  - A CAS assembly implementation is provided in the lab skeleton
- Use pthread synchronization to make several threads to preempt each other in order to play one ABA scenario
- Use a ABA-free performance test to compare performance of a lock-based and CAS-based concurrent stack
  - Warning: malloc/free operations expensive: make sure they don't occur in performance test
- Get more details and hints in the lab compendium

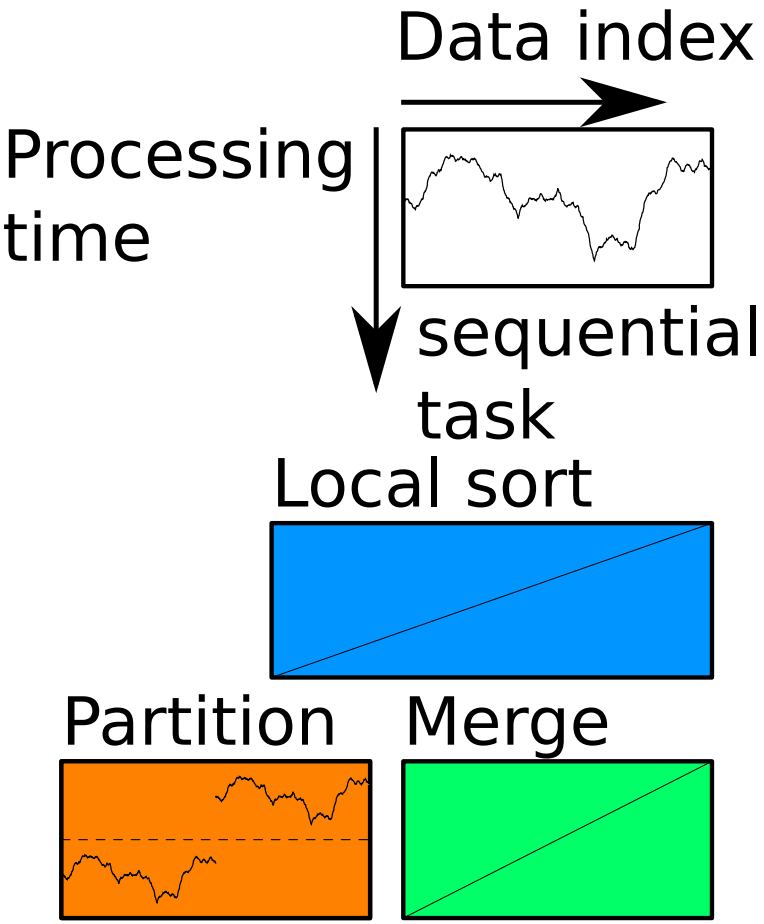
Lab 3: Parallel sorting

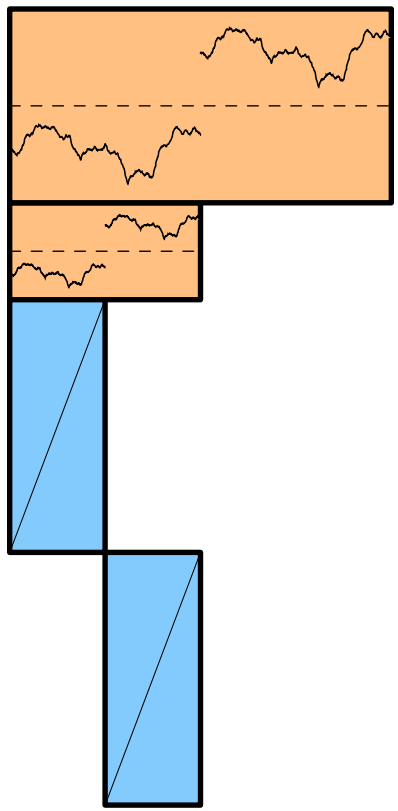
- Implement or optimize an existing sequential sort implementation
- Parallelize with shared memory approach (pthread or openMP)
- Parallelize with Streaming (Drake)
- Test your sorting implementation with various situations
  - Random, ascending, descending or constant input
  - Small and big input sizes
  - Other tricky situations you may imagine
- Built-in sorting functions (qsort(), std::sort()) are forbidden
  - May rewrite it for better performance
- Lab demo: describe the important techniques that accelerate your implementation.

Base sequential sort

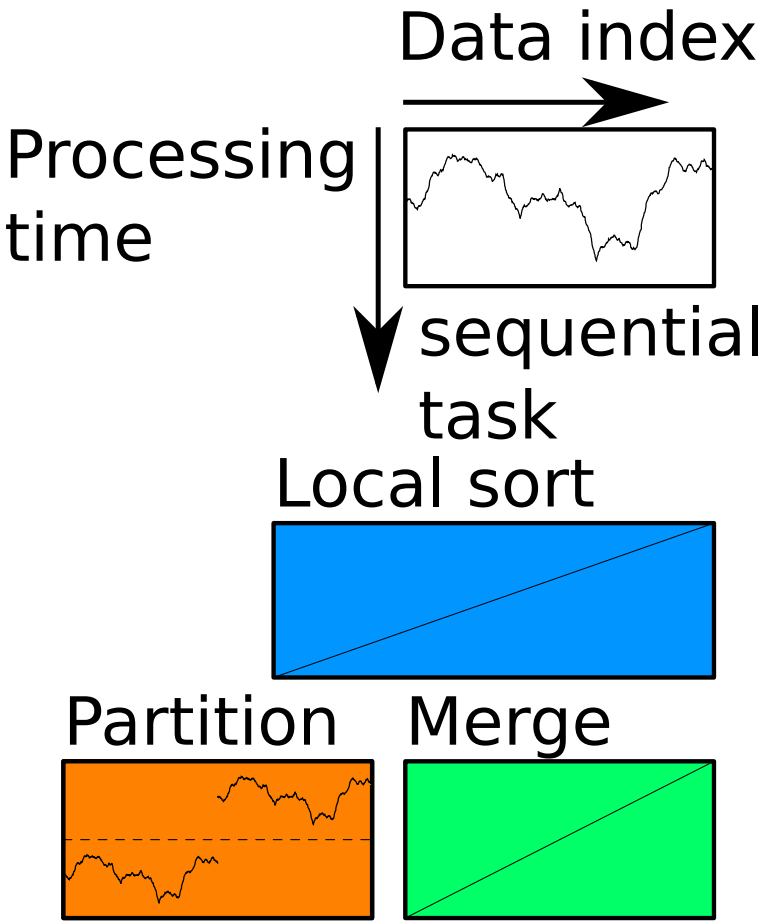
```
simple_quicksort(array, size) {
        pivot = array[size / 2];
        for(i = 0; i < size; i++) {}
                if(array[i] < pivot) {
                        left[left_size] = array[i];
                        left_size++;
                } else if(array[i] > pivot) {
                        right[right_size] = array[i];
                        right_size++;
                } else
                        pivot_count++;
        }
        simple_quicksort(left, left_size);
        simple_quicksort(right, right_size);
        memcpy(array, left, left_size * sizeof(int));
        for(i = left_size; i < left_size + pivot_count; i++)
                array[i] = pivot;
        memcpy(array + left_size + pivot_count, right, right_size * sizeof(int));
}
```

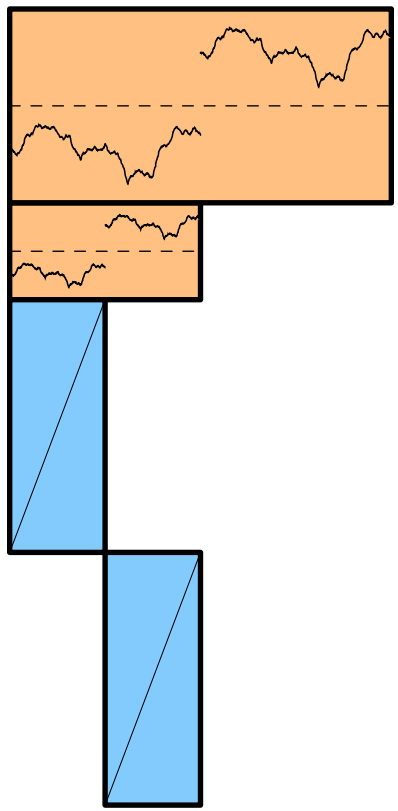
Base sequential sort

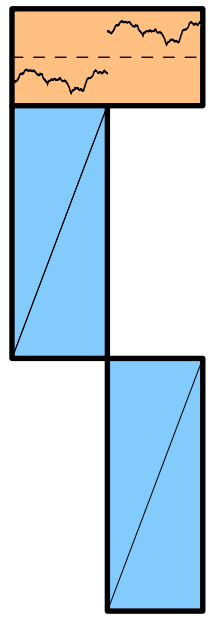




Base sequential sort



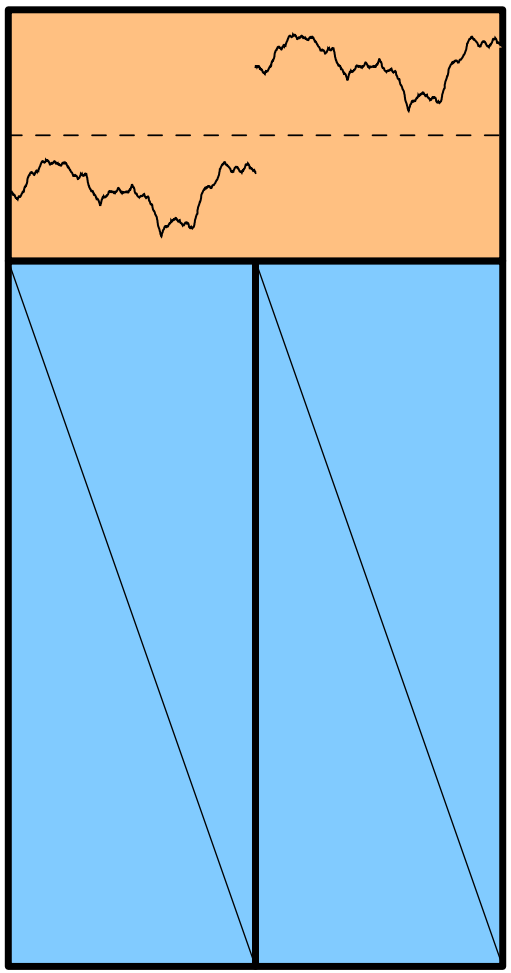




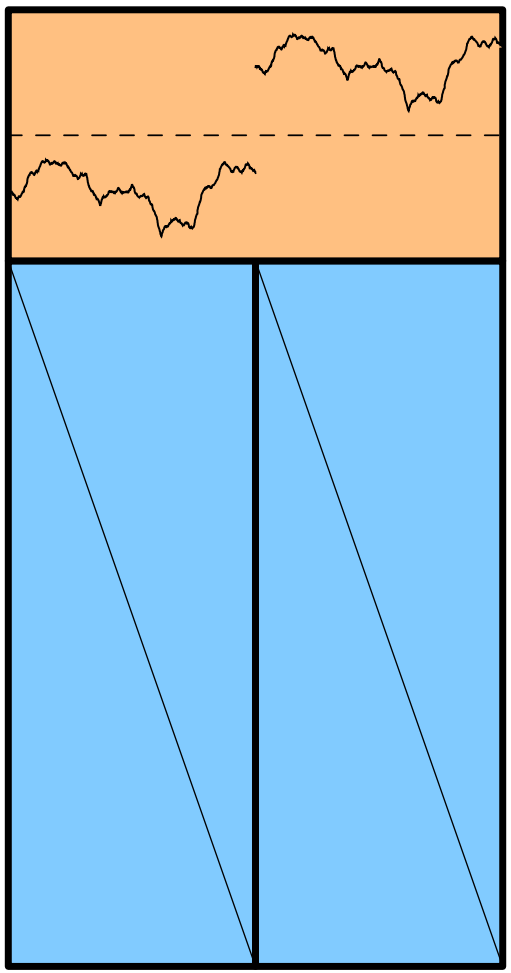
Parallelization opportunities

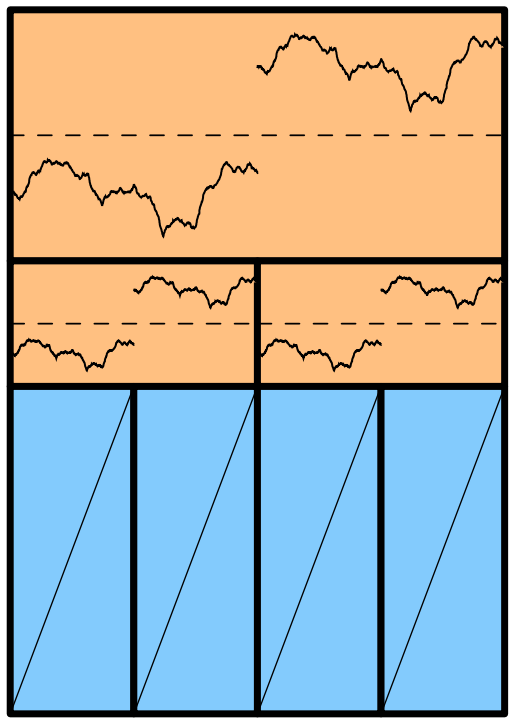
- Parallelization opportunities
  - Recursive calls
  - Computing pivots
  - Merging, if necessary
- Smart solutions challenging to implement
  - In-place quicksort: false sharing
  - Parallel sampling/merging: synchronization
  - Follow the KISS rule
- Avoid spawning more threads than the computer has cores
- Use data locality with caches and cache lines

Simple parallelization

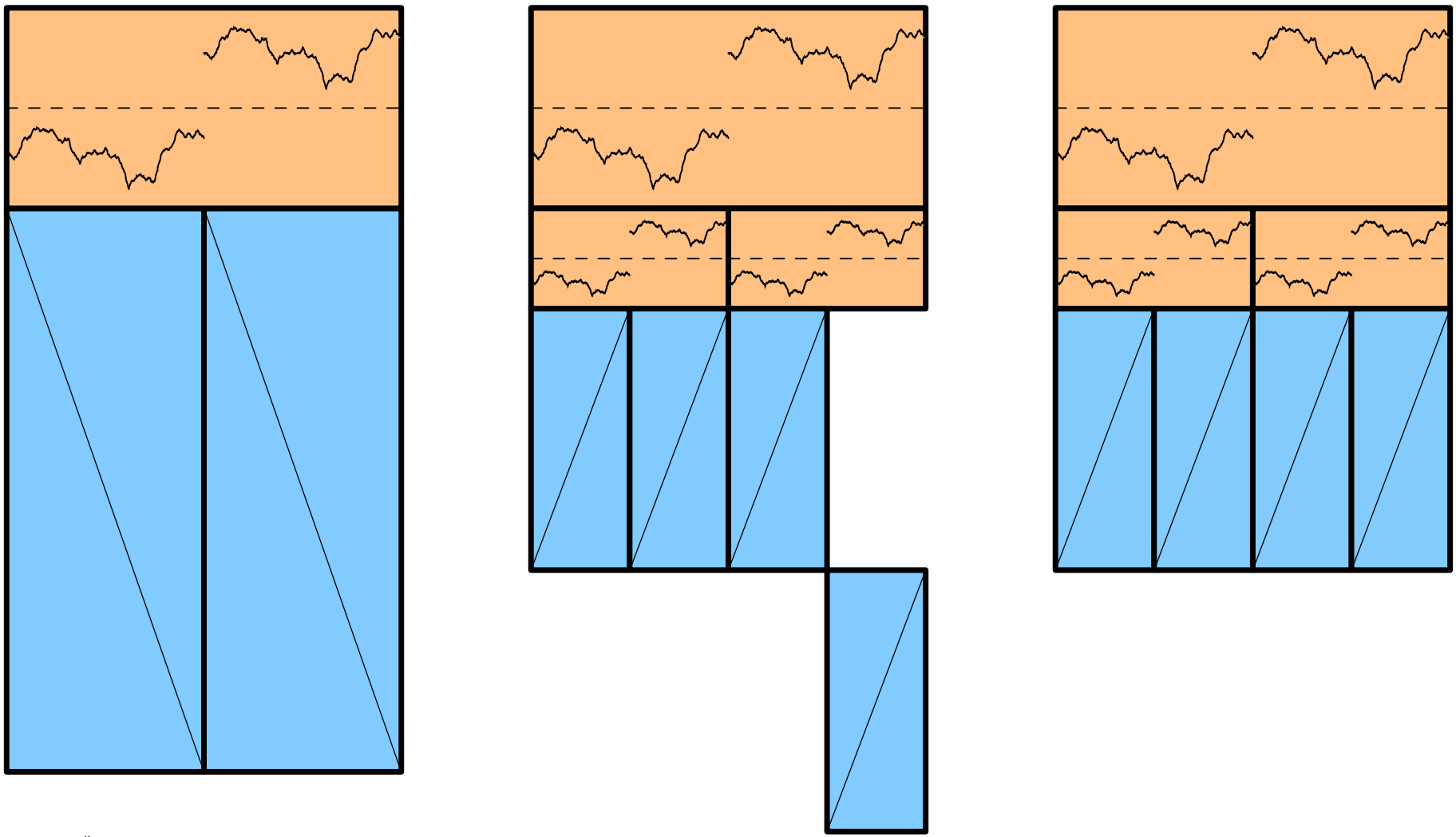


Simple parallelization





Simple parallelization



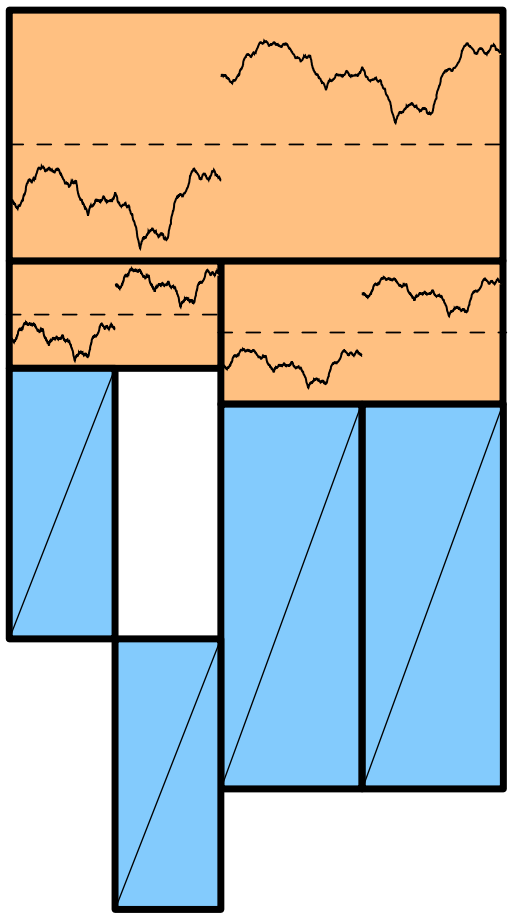
Parallel Quicksort sort with 3 cores

Can only efficiency use power of two number of cores.
How to use three cores efficiently?

Parallel Quicksort sort with 3 cores

Can only efficiency use power of two number of cores.
How to use three cores efficiently?

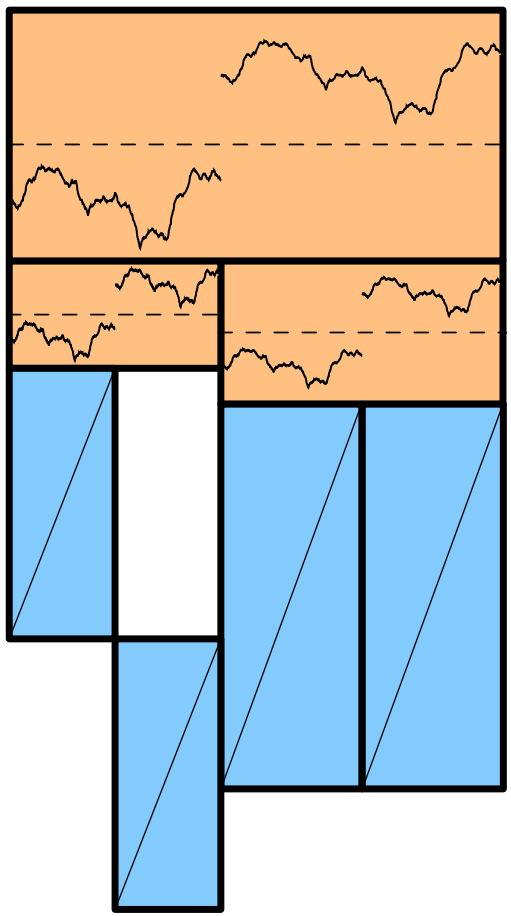
- Choose pivot to divide buffer into unequal parts

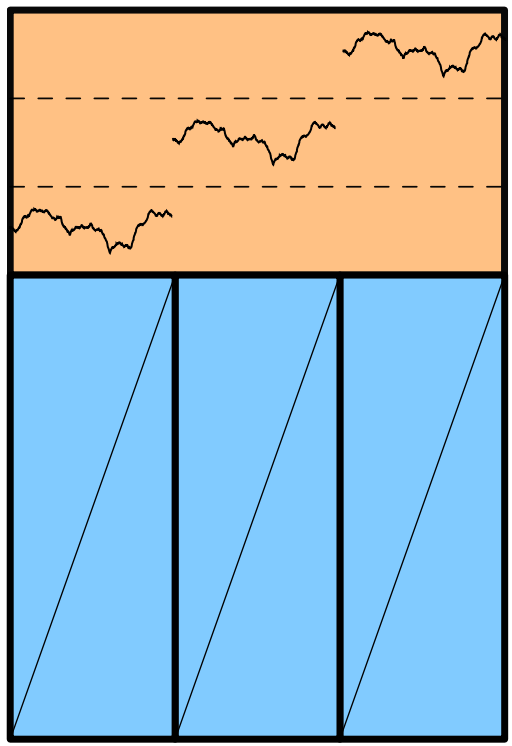


Parallel Quicksort sort with 3 cores

Can only efficiency use power of two number of cores.
How to use three cores efficiently?

- Choose pivot to divide buffer into unequal parts
- Partition and recurse into 3 parts (sample sort)

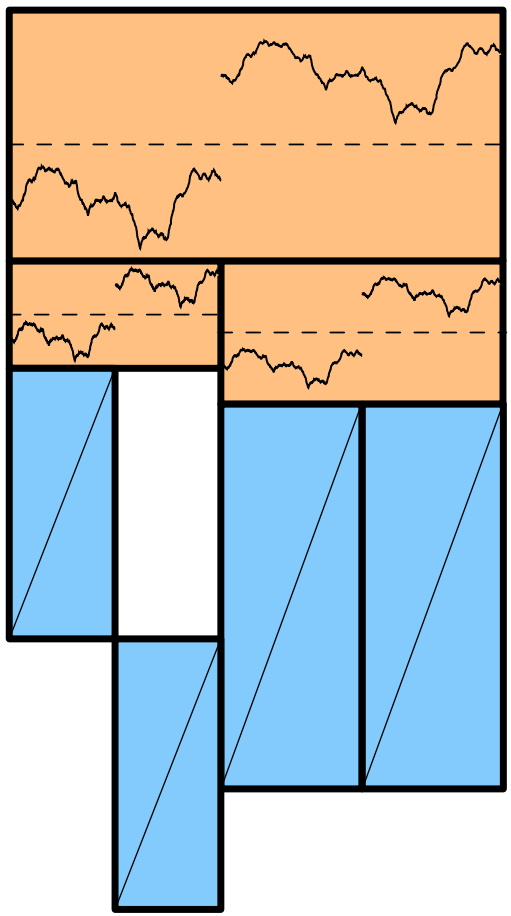


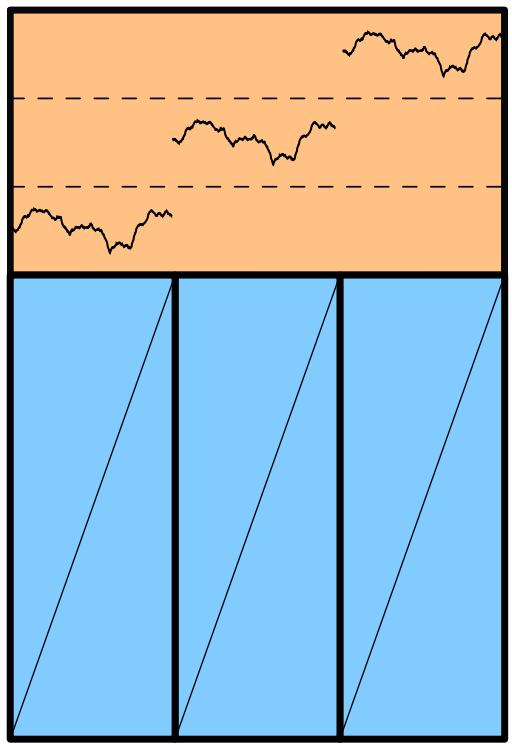


Parallel Quicksort sort with 3 cores

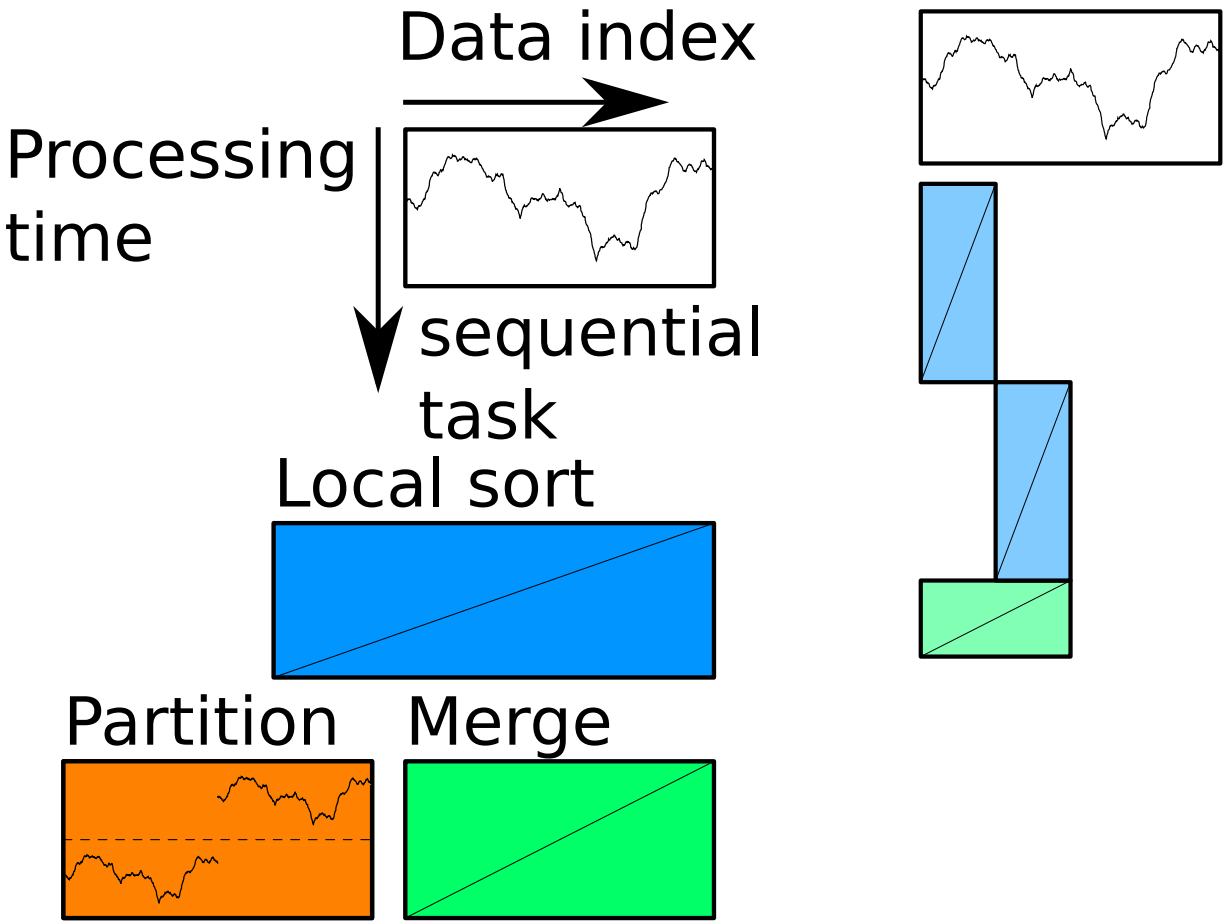
Can only efficiency use power of two number of cores.
How to use three cores efficiently?

- Choose pivot to divide buffer into unequal parts
- Partition and recurse into 3 parts (sample sort)
- Makes implementation harder

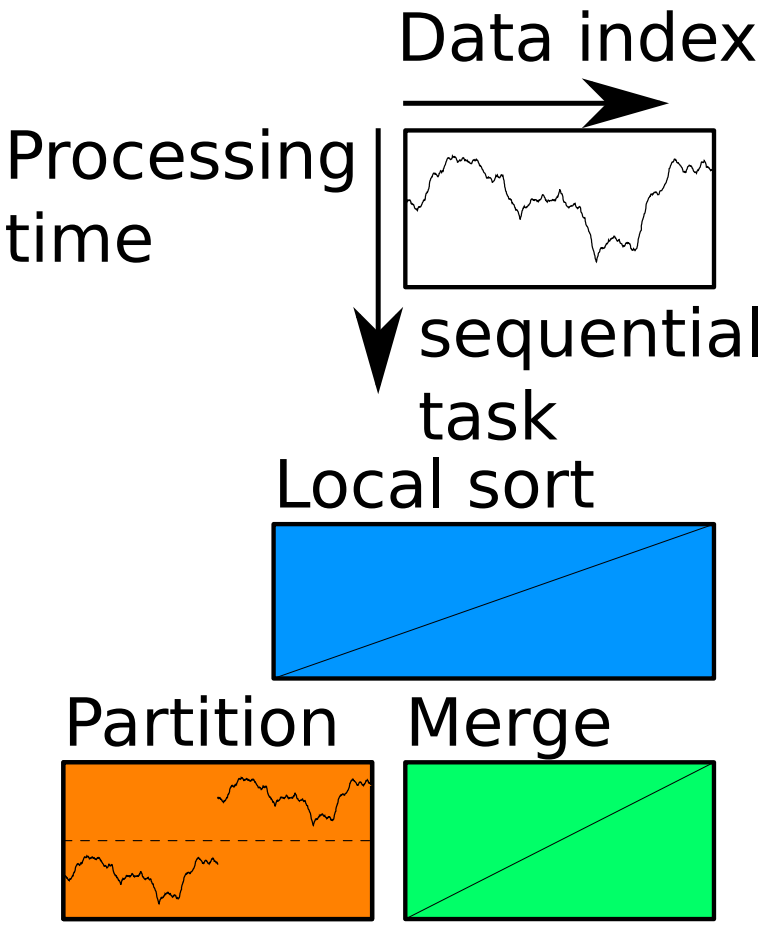


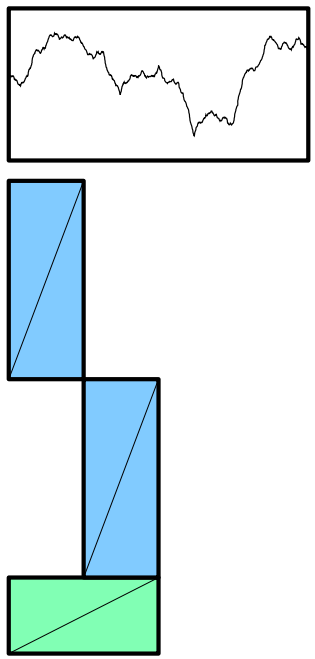


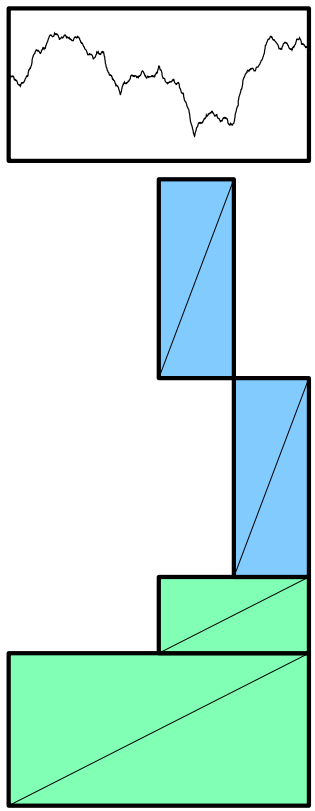
Mergesort


Data index
Processing time
sequential task
Local sort
Partition
Merge

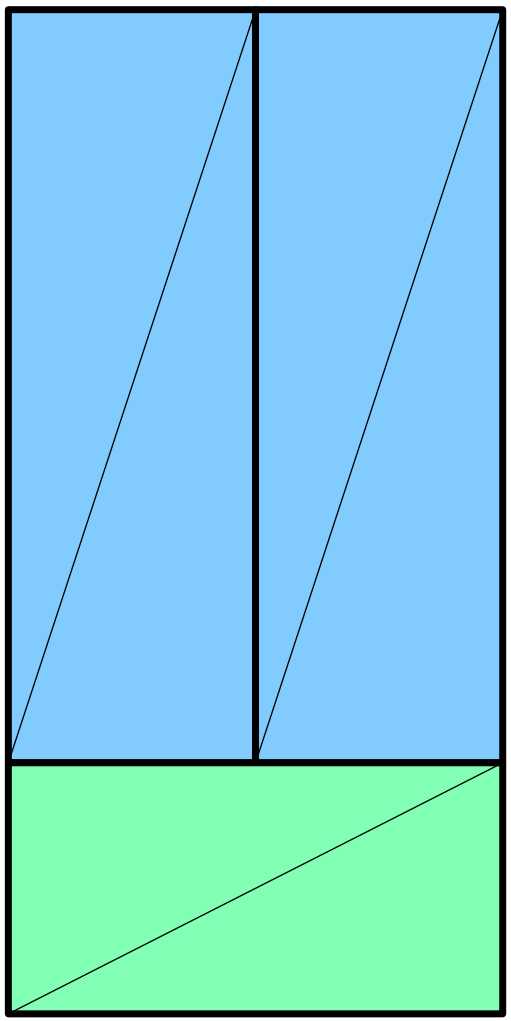
Mergesort


Data index
Processing time
sequential task
Local sort
Partition
Merge

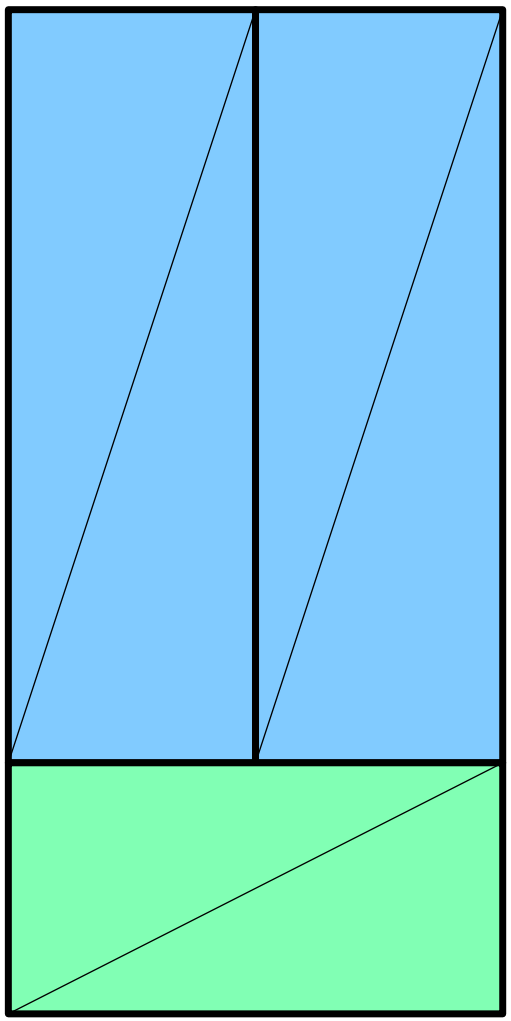


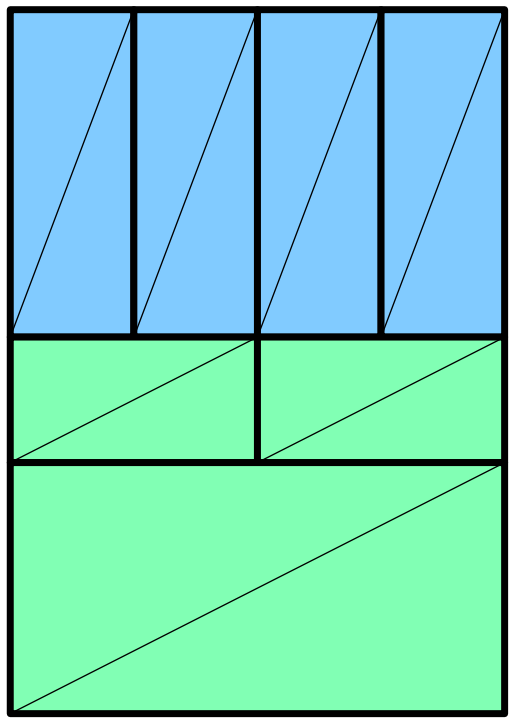


Simple Mergesort Parallelization



Simple Mergesort Parallelization





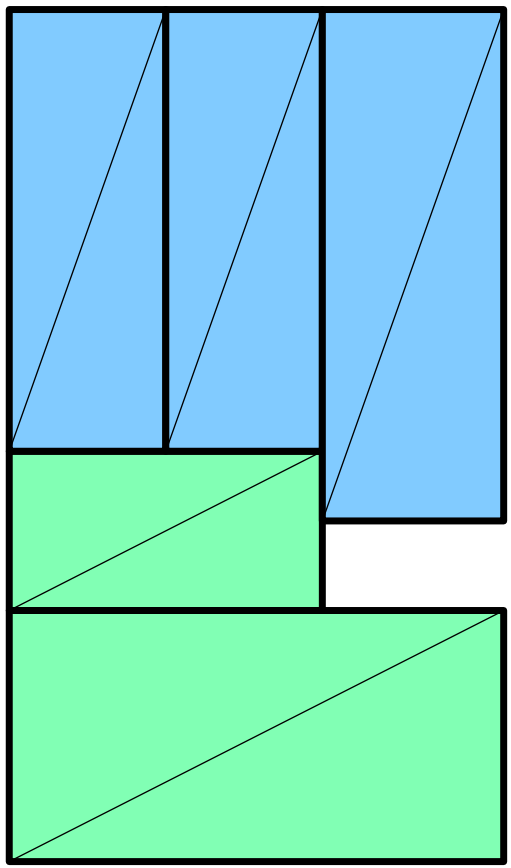
Parallel Mergesort with 3 cores

Can only efficiency use power of two number of cores.
How to use three cores efficiently?

Parallel Mergesort with 3 cores

Can only efficiency use power of two number of cores.
How to use three cores efficiently?

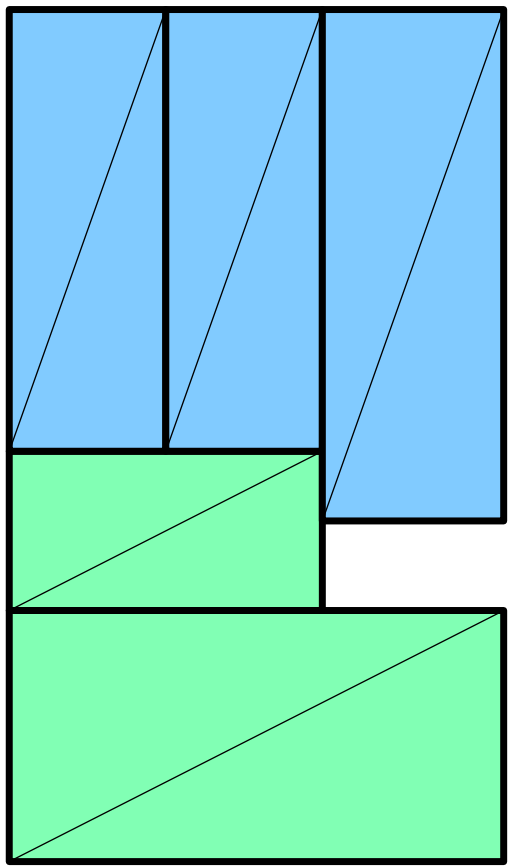
- Divide buffer into 2 unequal parts

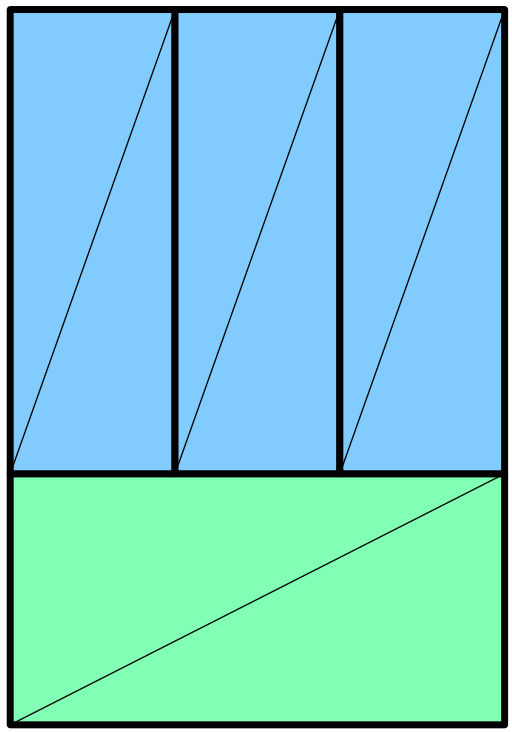


Parallel Mergesort with 3 cores

Can only efficiency use power of two number of cores.
How to use three cores efficiently?

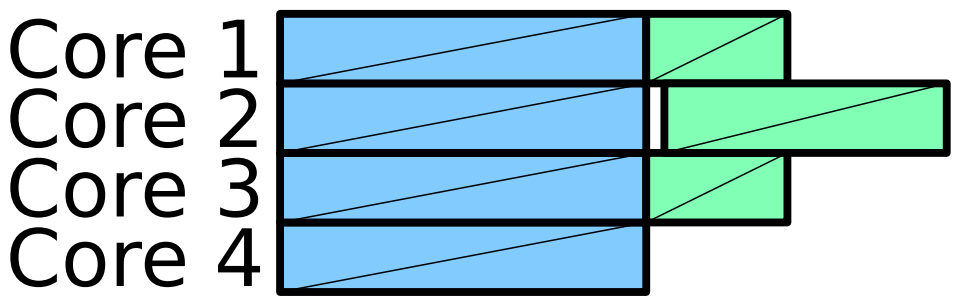
- Divide buffer into 2 unequal parts
- Partition and recurse into 3 parts and 3-way merge





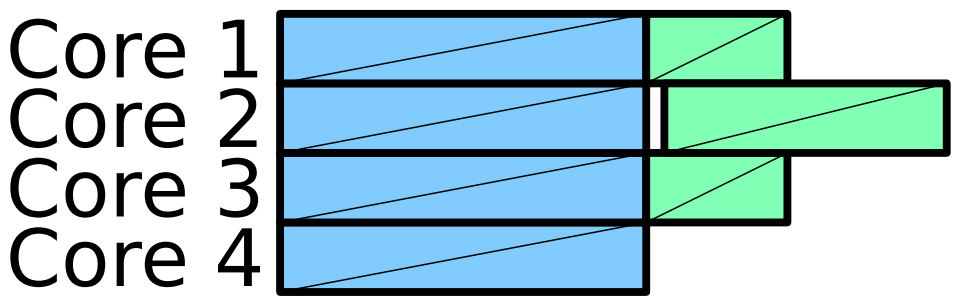
Pipelined parallel mergesort

Classic parallelism: start a task when the previous one is done Pipeline parallelism: Run next merging task as soon as possible



Pipelined parallel mergesort

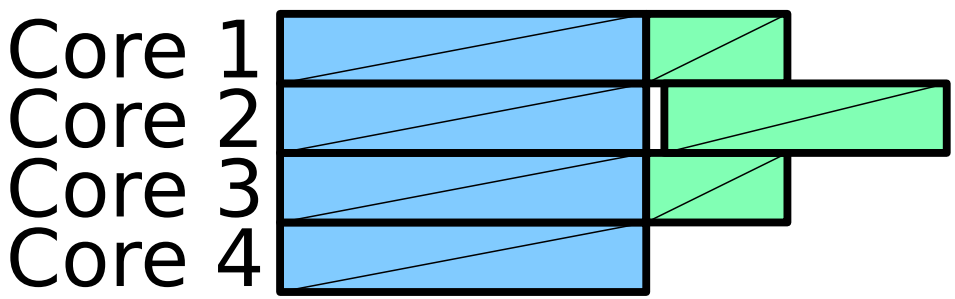
Classic parallelism: start a task when the previous one is done Pipeline parallelism: Run next merging task as soon as possible



- Even more speedup

Pipelined parallel mergesort

Classic parallelism: start a task when the previous one is done Pipeline parallelism: Run next merging task as soon as possible



- Even more speedup
- Difficult to implement manually

Pipeline parallelism

Related research since the 60’

- Program verifiability

Pipeline parallelism

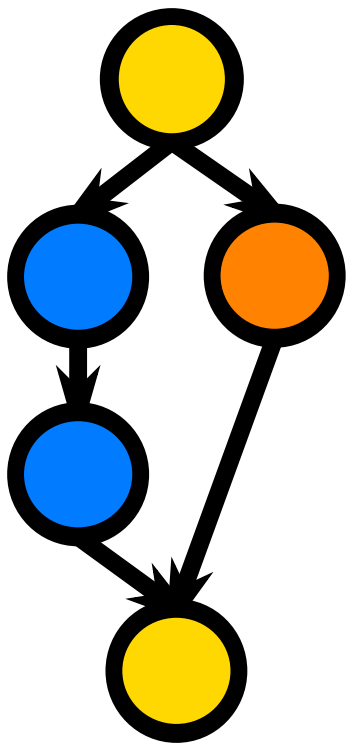
Related research since the 60'

- Program verifiability
- Parallelism is a mere "consequence"

Pipeline parallelism

Related research since the 60'

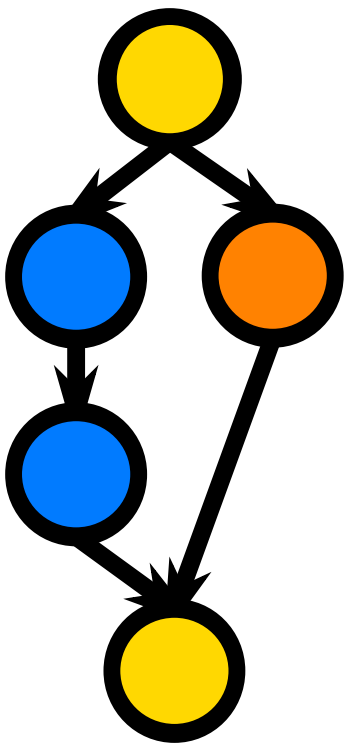
- Program verifiability
- Parallelism is a mere "consequence"
- Sequential tasks communicating through channels

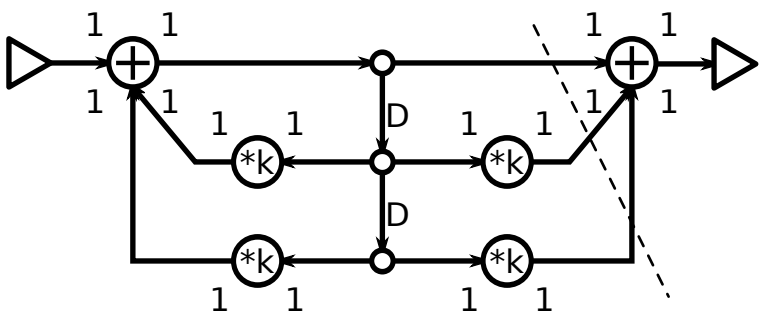


Pipeline parallelism

Related research since the 60'

- Program verifiability
- Parallelism is a mere "consequence"
- Sequential tasks communicating through channels
- Theories: Kahn Networks, (Synchronous) Data Flow, Communicating Sequential Processes

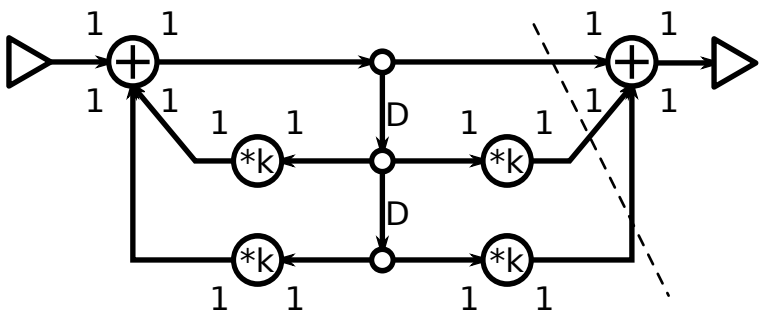


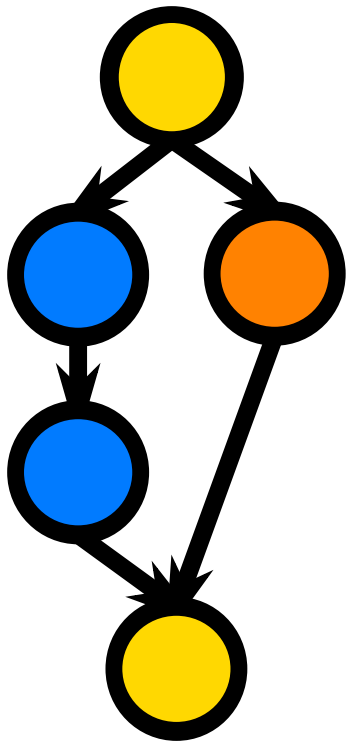


Pipeline parallelism

Related research since the 60'

- Program verifiability
- Parallelism is a mere "consequence"
- Sequential tasks communicating through channels
- Theories: Kahn Networks, (Synchronous) Data Flow, Communicating Sequential Processes
- Languages: Streamit, CAL, Esterel





Classic versus stream

Most programming languages unsuitable to parallelism

- Single, universal instruction pointer
- Single, universal address space

Classic versus stream

Most programming languages unsuitable to parallelism

- Single, universal instruction pointer
- Single, universal address space
- Difficult to read with several threads in mind

Classic versus stream

Most programming languages unsuitable to parallelism

- Single, universal instruction pointer
- Single, universal address space
- Difficult to read with several threads in mind
- Annotations (OpenMP) not helping with high number of cores

Classic versus stream

Most programming languages unsuitable to parallelism

- Single, universal instruction pointer
- Single, universal address space
- Difficult to read with several threads in mind
- Annotations (OpenMP) not helping with high number of cores

Stream programming

- (Mostly) sequential tasks

Classic versus stream

Most programming languages unsuitable to parallelism

- Single, universal instruction pointer
- Single, universal address space
- Difficult to read with several threads in mind
- Annotations (OpenMP) not helping with high number of cores

Stream programming

- (Mostly) sequential tasks
- Actual parallelism: scheduling

Classic versus stream

Most programming languages unsuitable to parallelism

- Single, universal instruction pointer
- Single, universal address space
- Difficult to read with several threads in mind
- Annotations (OpenMP) not helping with high number of cores

Stream programming

- (Mostly) sequential tasks
- Actual parallelism: scheduling
- No universal shared memory

Classic versus stream

Most programming languages unsuitable to parallelism

- Single, universal instruction pointer
- Single, universal address space
- Difficult to read with several threads in mind
- Annotations (OpenMP) not helping with high number of cores

Stream programming

- (Mostly) sequential tasks
- Actual parallelism: scheduling
- No universal shared memory
- Natural to pipeline parallelism

Classic versus stream

Most programming languages unsuitable to parallelism

- Single, universal instruction pointer
- Single, universal address space
- Difficult to read with several threads in mind
- Annotations (OpenMP) not helping with high number of cores

Stream programming

- (Mostly) sequential tasks
- Actual parallelism: scheduling
- No universal shared memory
- Natural to pipeline parallelism
- Communications with on-chip memories: on-chip pipelining

Scheduling

Optimize distribute tasks to cores

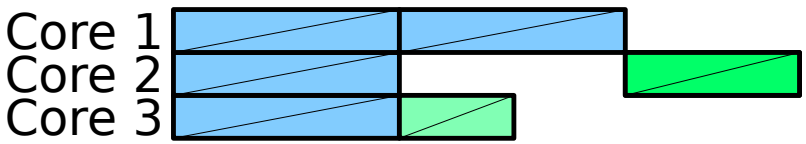
- For speed, energy, power or a compromise
- Sequential, moldable or malleable tasks
- Power management: DVFS, Sleeping states

Abundant research

- Longest Processing Time First (LPT)
- Crown Scheduling

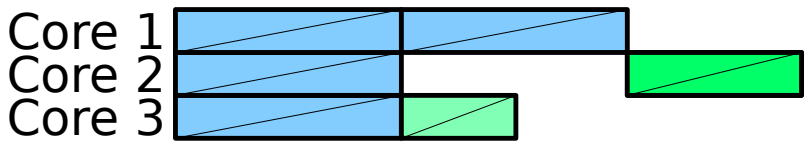
Back to parallel merge

Classic parallelism

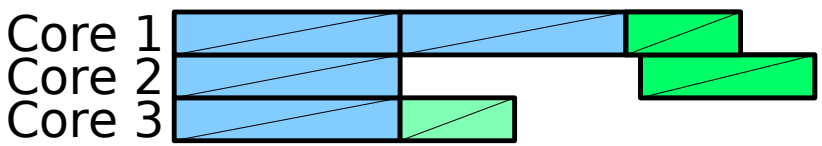


Back to parallel merge

Classic parallelism

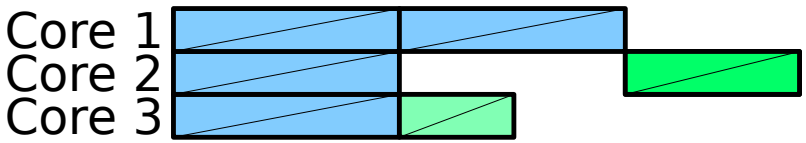


Pipelining (4 initial sorting tasks)

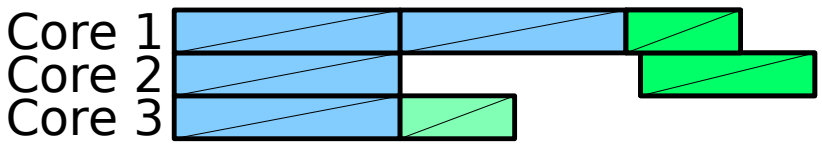


Back to parallel merge

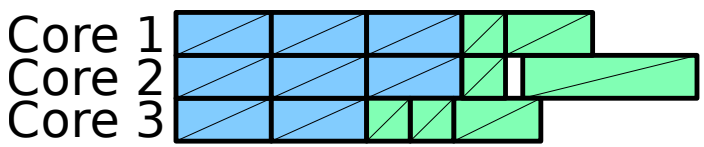
Classic parallelism



Pipelining (4 initial sorting tasks)

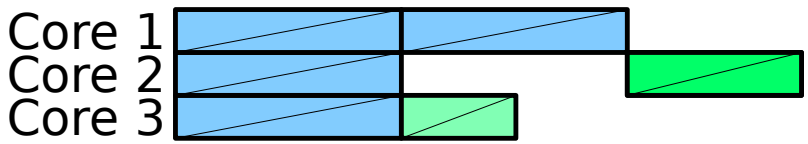


Pipelining (8 initial sorting tasks)

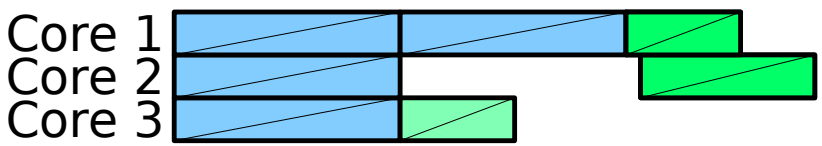


Back to parallel merge

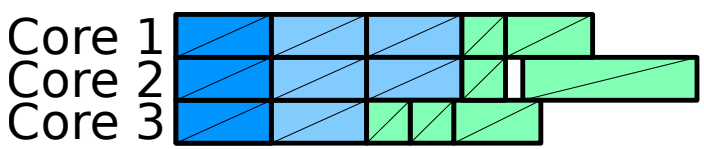
Classic parallelism



Pipelining (4 initial sorting tasks)

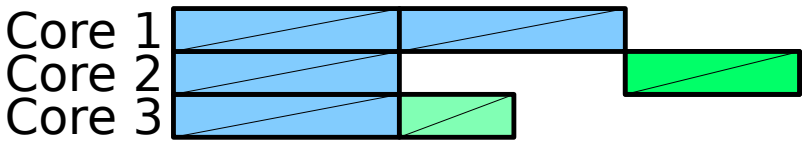


Pipelining (8 initial sorting tasks)

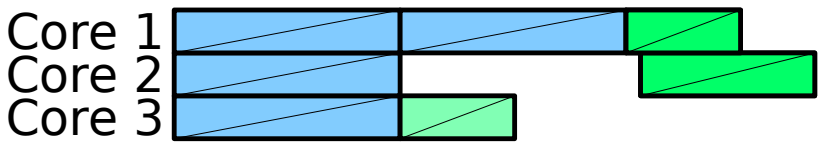


Back to parallel merge

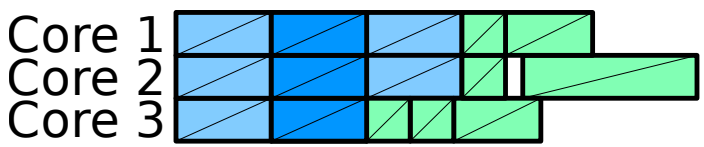
Classic parallelism



Pipelining (4 initial sorting tasks)

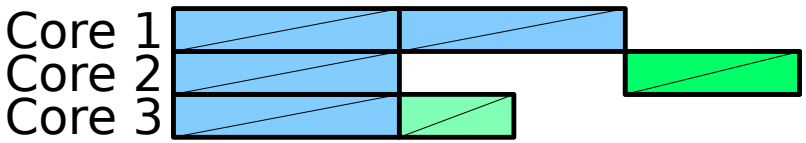


Pipelining (8 initial sorting tasks)

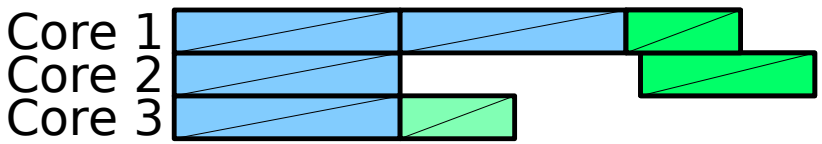


Back to parallel merge

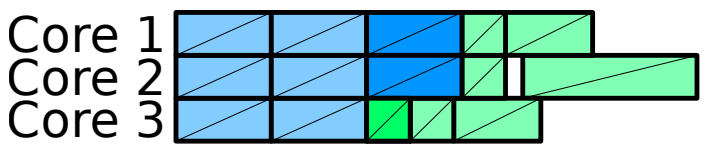
Classic parallelism



Pipelining (4 initial sorting tasks)

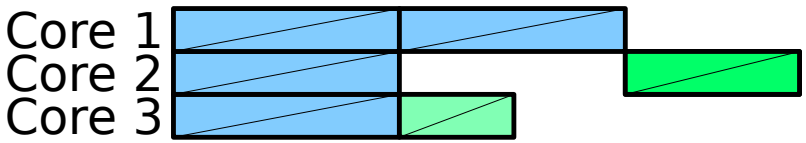


Pipelining (8 initial sorting tasks)

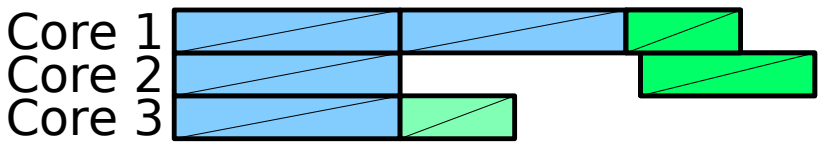


Back to parallel merge

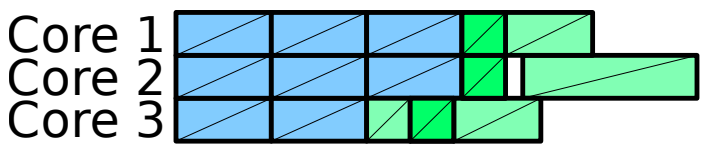
Classic parallelism



Pipelining (4 initial sorting tasks)

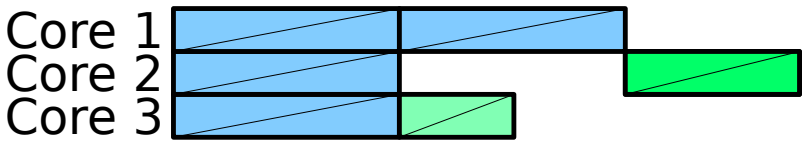


Pipelining (8 initial sorting tasks)

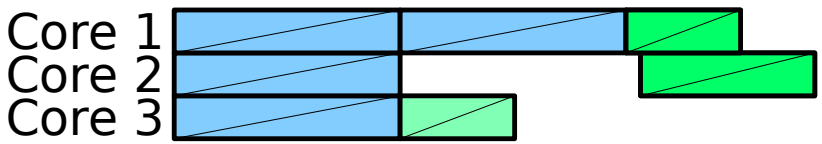


Back to parallel merge

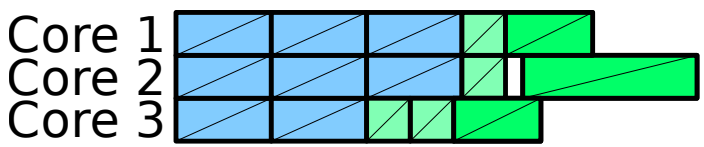
Classic parallelism



Pipelining (4 initial sorting tasks)



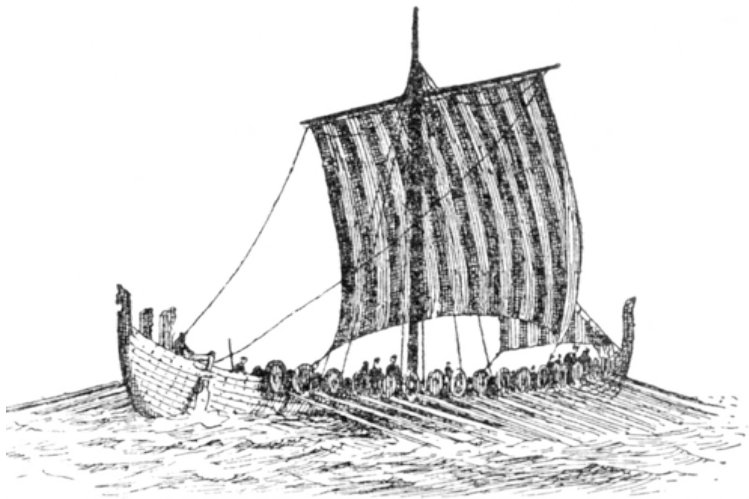
Pipelining (8 initial sorting tasks)

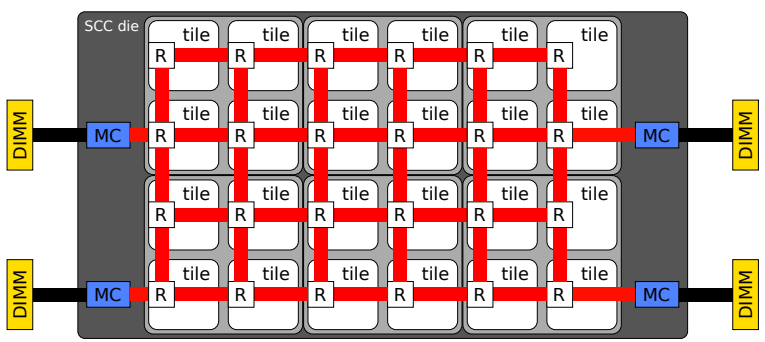


Drake

Drake: A C framework for Streaming Applications

- Separate design
  - Application
  - Platform

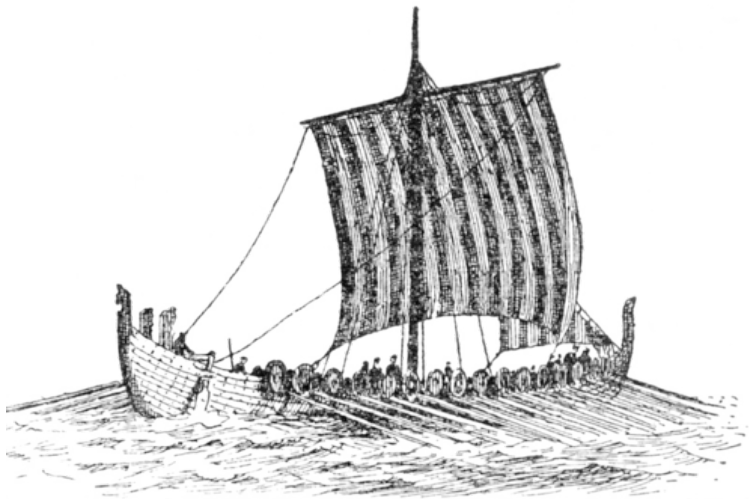


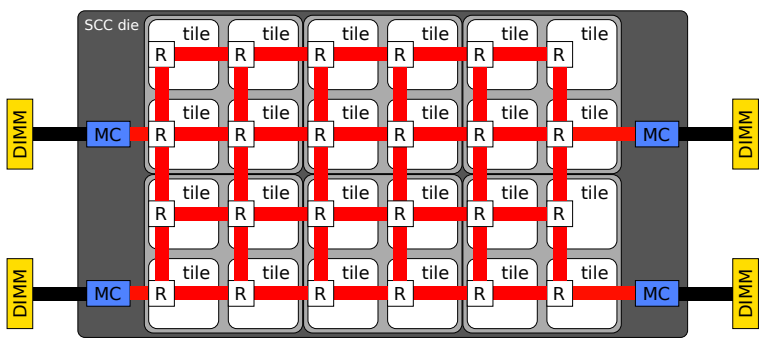


Drake

Drake: A C framework for Streaming Applications

- Separate design
  - Application
  - Platform
- Parallelism with scheduling

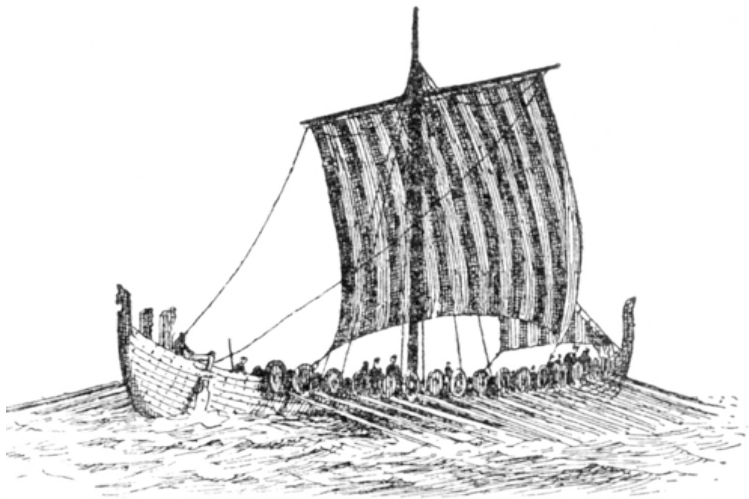


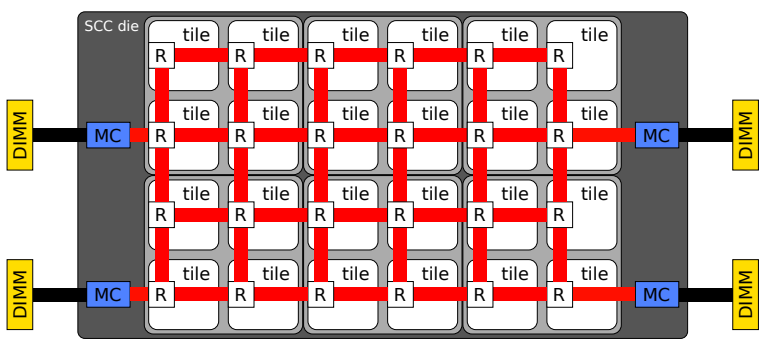


Drake

Drake: A C framework for Streaming Applications

- Separate design
  - Application
  - Platform
- Parallelism with scheduling
- Energy efficiency
  - Moldable tasks
  - Frequency scaling
  - Power management

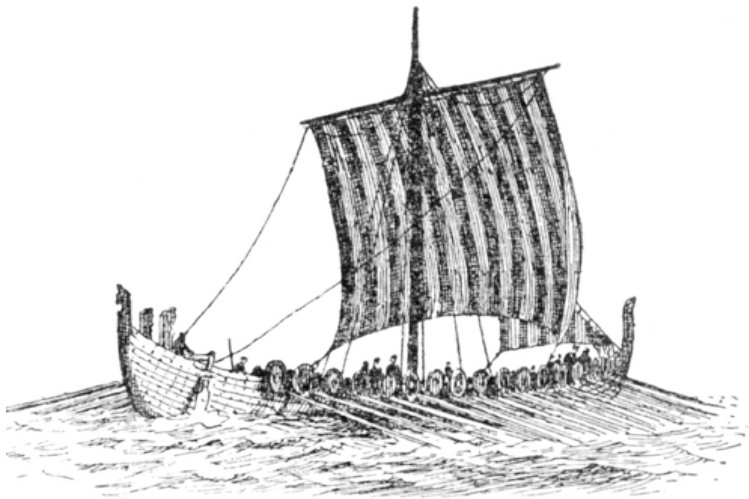


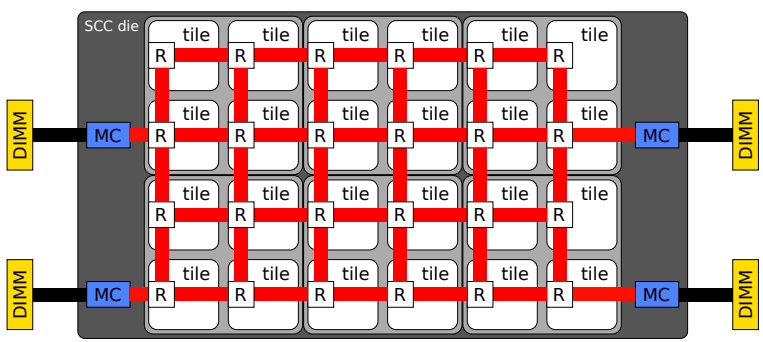


Drake

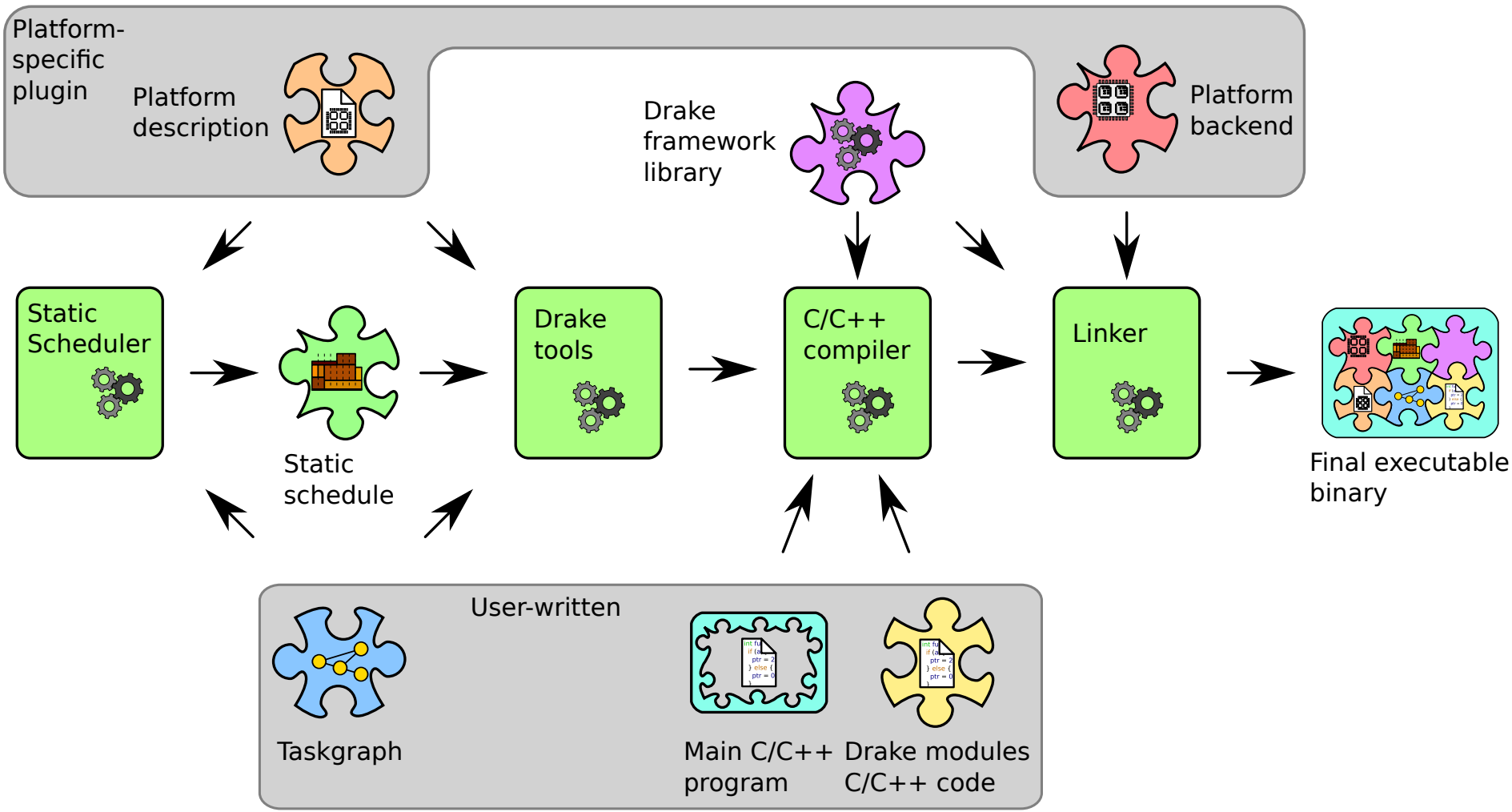
Drake: A C framework for Streaming Applications

- Separate design
  - Application
  - Platform
- Parallelism with scheduling
- Energy efficiency
  - Moldable tasks
  - Frequency scaling
  - Power management
- Multi target
  - Intel SCC
  - Intel Xeon





Design with Drake



Lab 3: Directions

Part 1: Classic parallel sort

- Parallelize the sequential sort provided in src/sort.cpp. Keep it simple
- Optimize it so it can use 3 cores efficiently

Lab 3: Directions

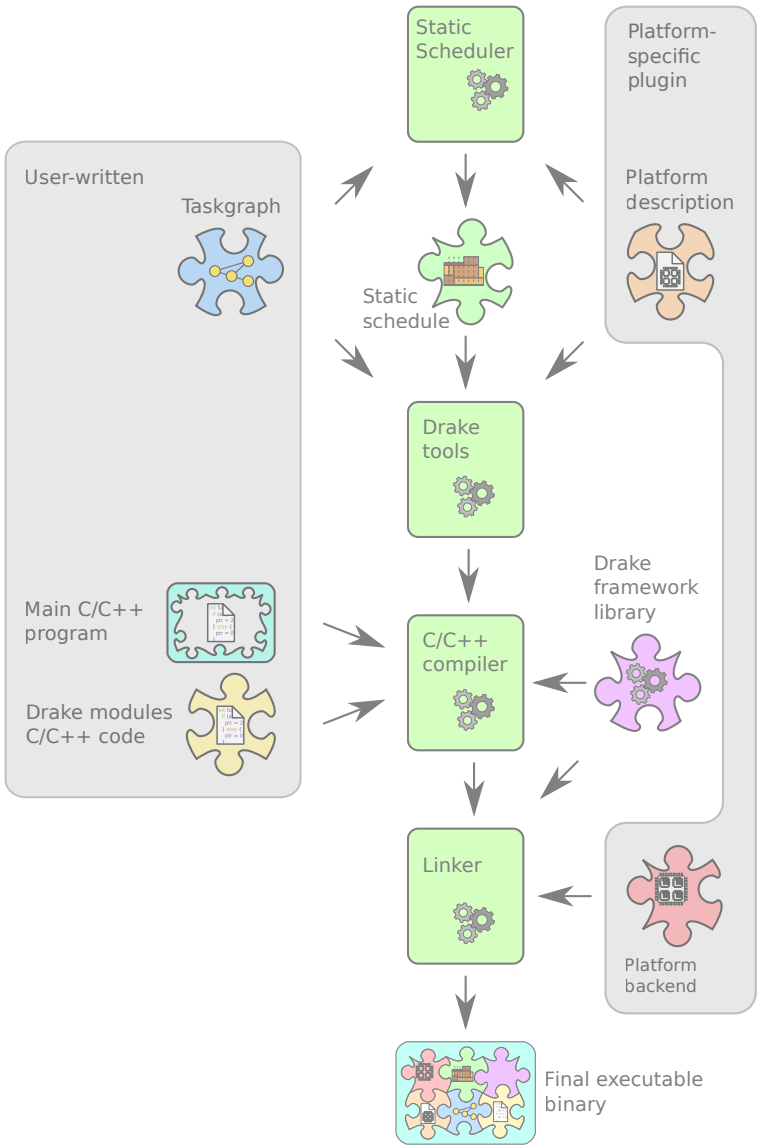
Part 1: Classic parallel sort

- Parallelize the sequential sort provided in src/sort.cpp. Keep it simple
- Optimize it so it can use 3 cores efficiently

Part 2: Pipelined parallel mergesort using Drake

Lab3: Directions

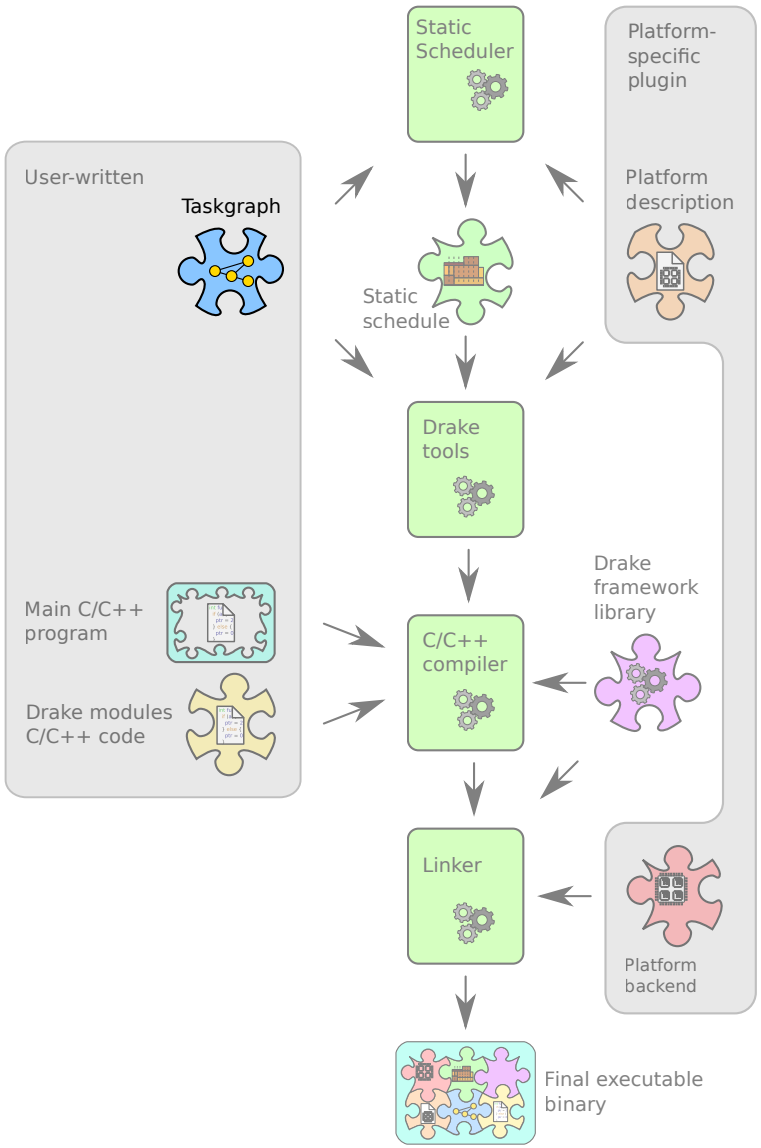
Development steps



Lab3: Directions

Development steps

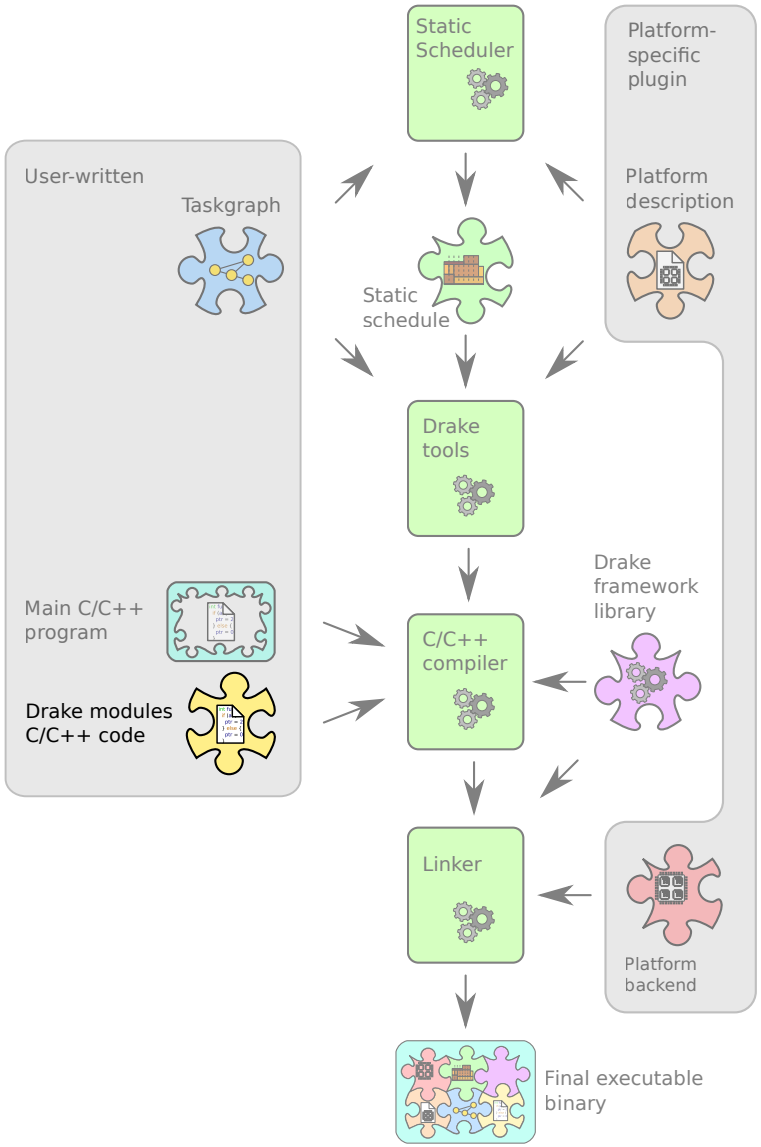
1. Describe streaming application
   - *src/merge*.graphml*



Lab3: Directions

Development steps

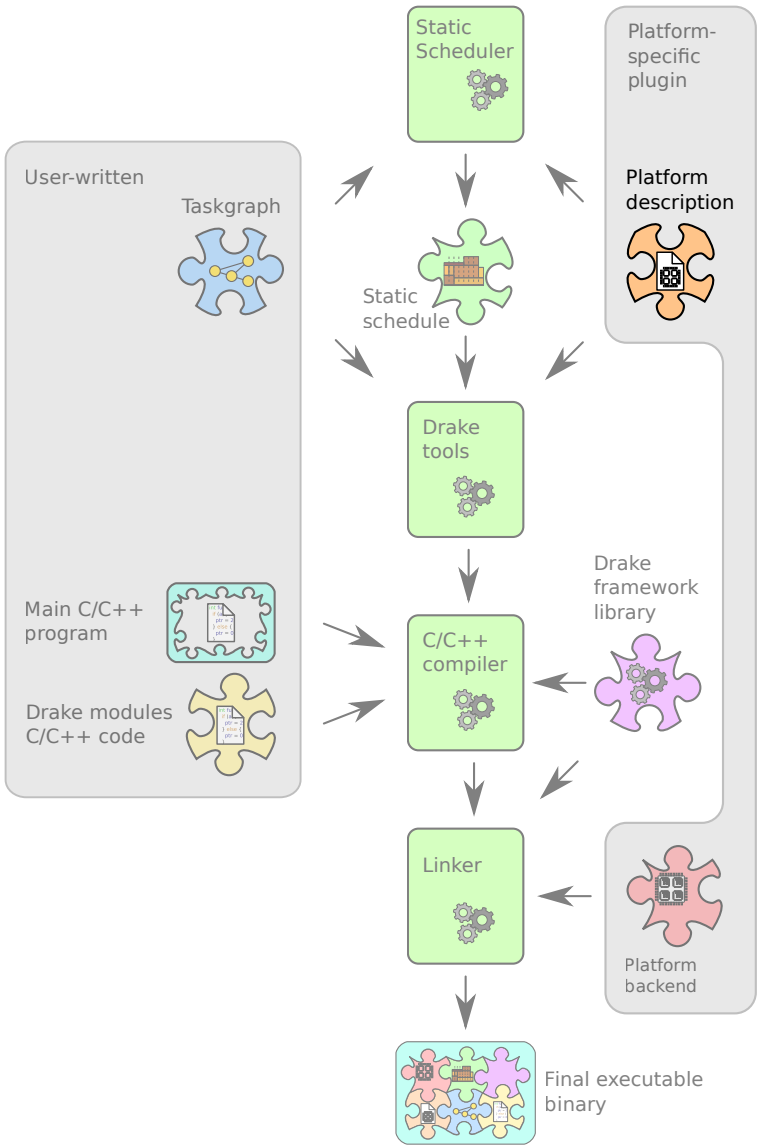
1. Describe streaming application
   - *src/merge*.graphml*
2. Write task code
   - *src/merge.c*



Lab3: Directions

Development steps

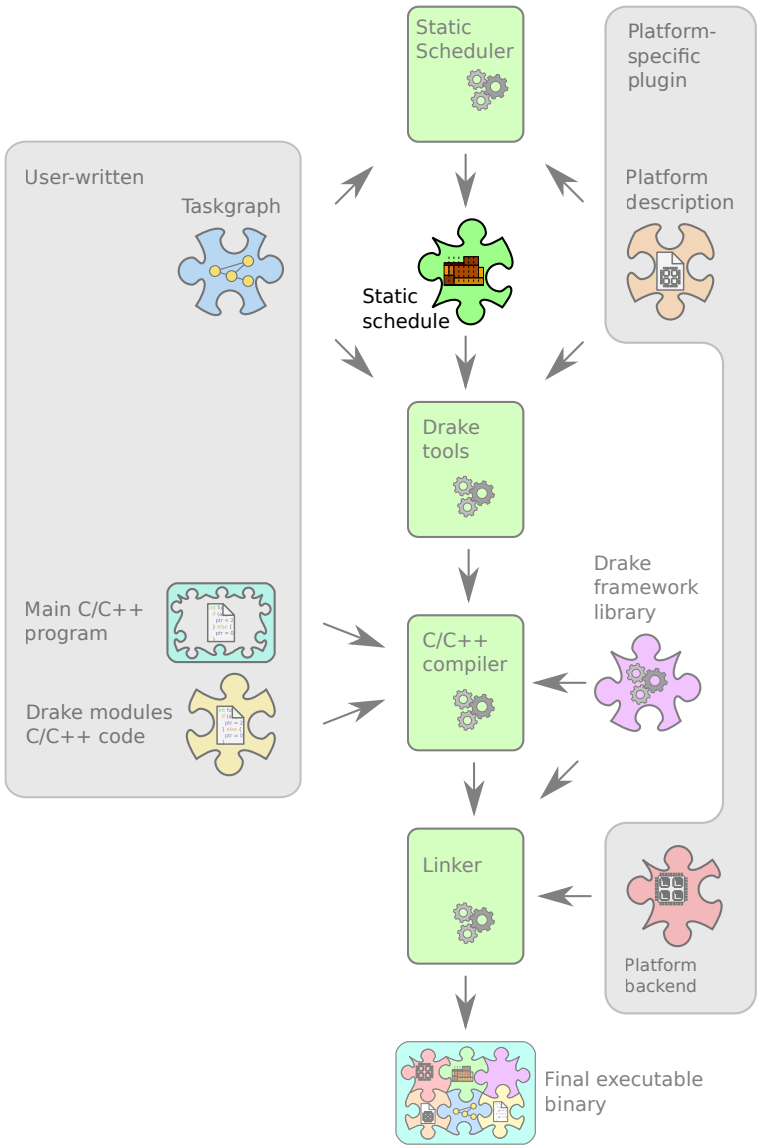
1. Describe streaming application
   - *src/merge*.graphml*
2. Write task code
   - *src/merge.c*
3. Describe target platform
   - *src/platform*.dat*



Lab3: Directions

Development steps

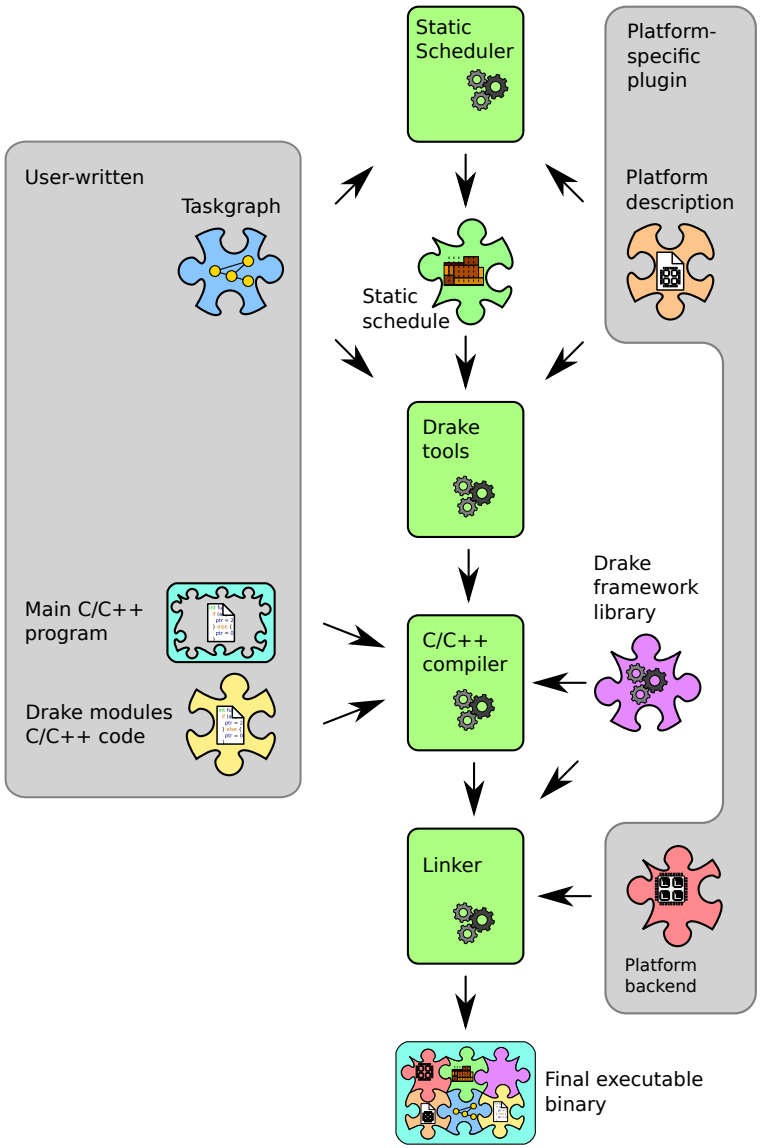
1. Describe streaming application
   - *src/merge*.graphml*
2. Write task code
   - *src/merge.c*
3. Describe target platform
   - *src/platform*.dat*
4. Define schedule
   - *src/schedule*.dat*



Lab3: Directions

Development steps

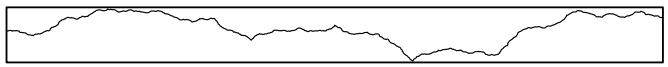
1. Describe streaming application
   - *src/merge*.graphml*
2. Write task code
   - *src/merge.c*
3. Describe target platform
   - *src/platform*.dat*
4. Define schedule
   - *src/schedule*.dat*
5. Compile application

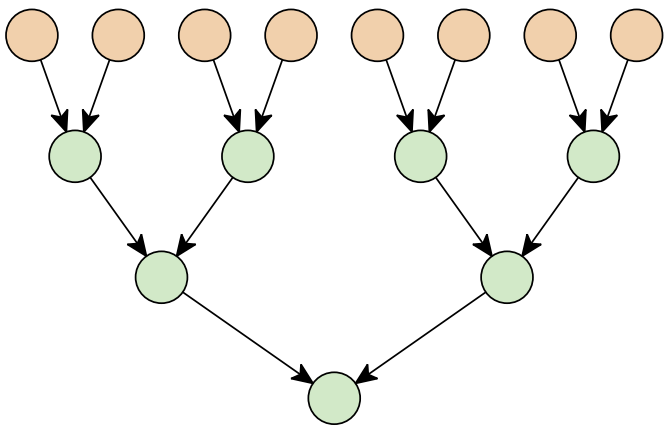


Describe Streaming Application

Describe Mergesort

- Split program into tasks

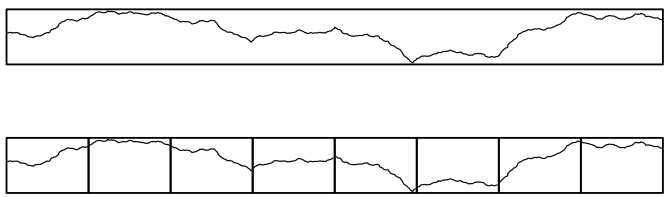


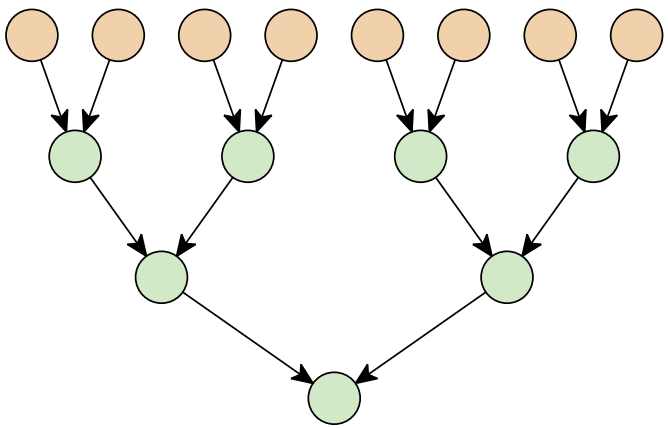


Describe Streaming Application

Describe Mergesort

- Split program into tasks
  - Partitioning

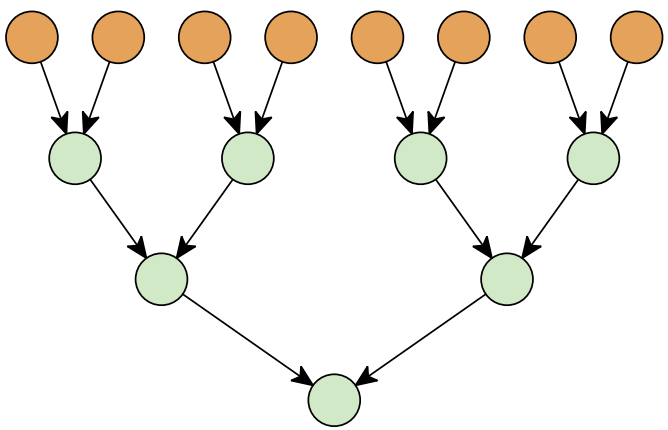


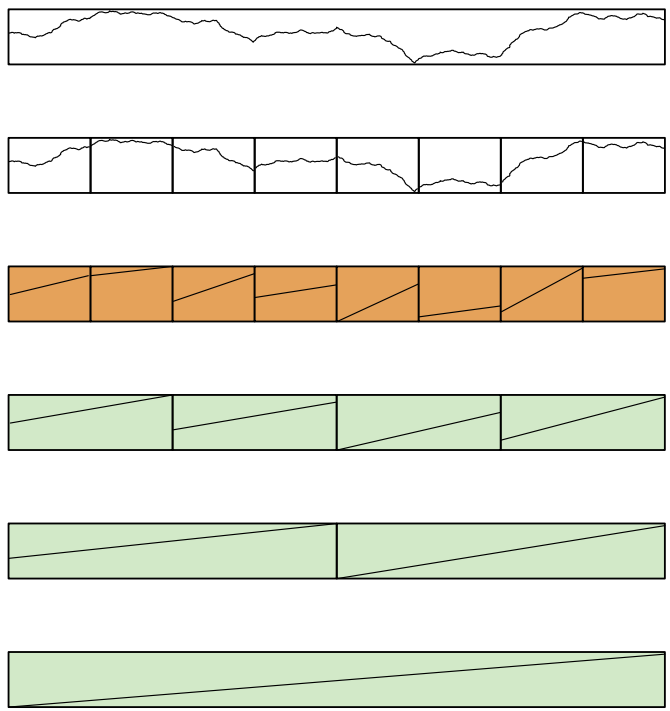


Describe Streaming Application

Describe Mergesort

- Split program into tasks
  - Partitioning
  - Presorting

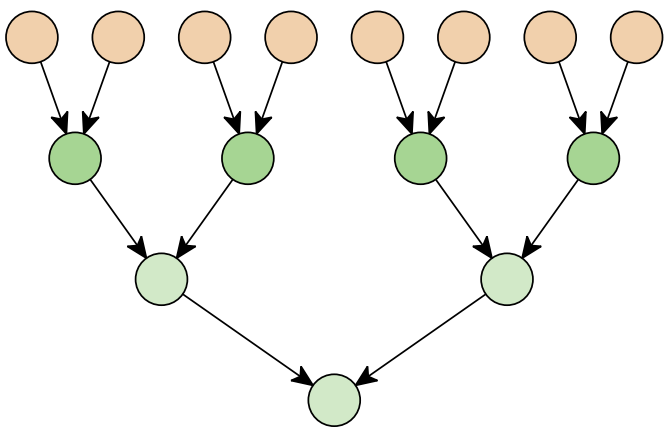


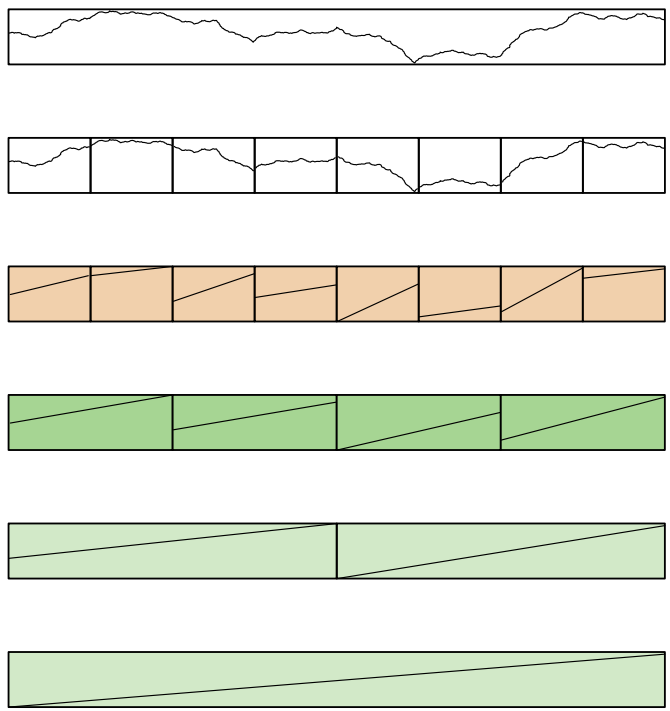


Describe Streaming Application

Describe Mergesort

- Split program into tasks
  - Partitioning
  - Presorting
  - Merging

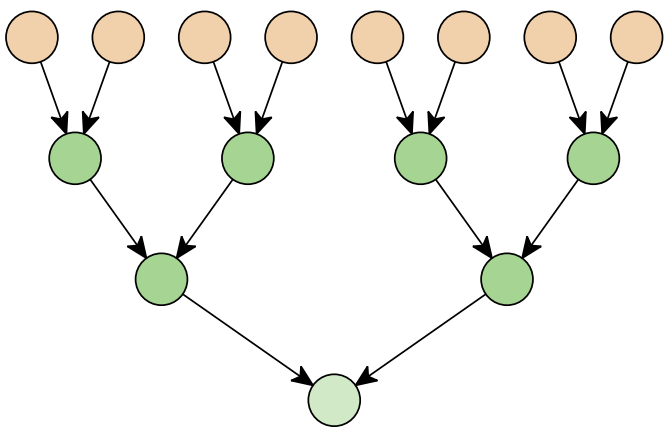


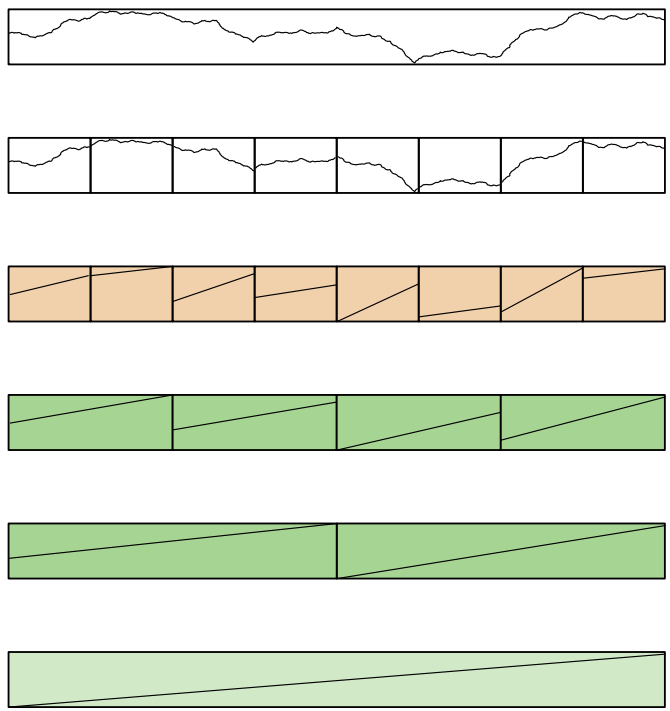


Describe Streaming Application

Describe Mergesort

- Split program into tasks
  - Partitioning
  - Presorting
  - Merging

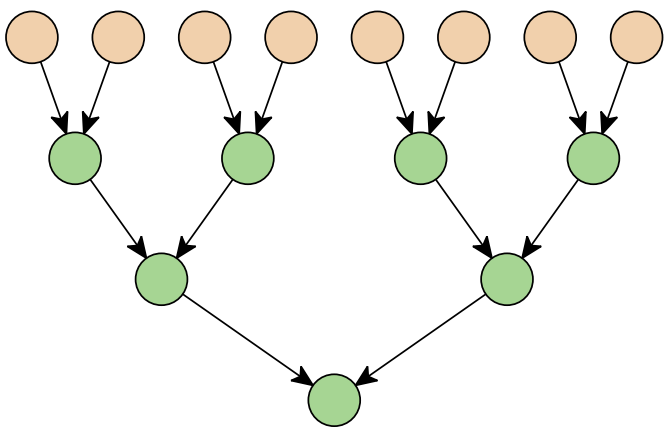


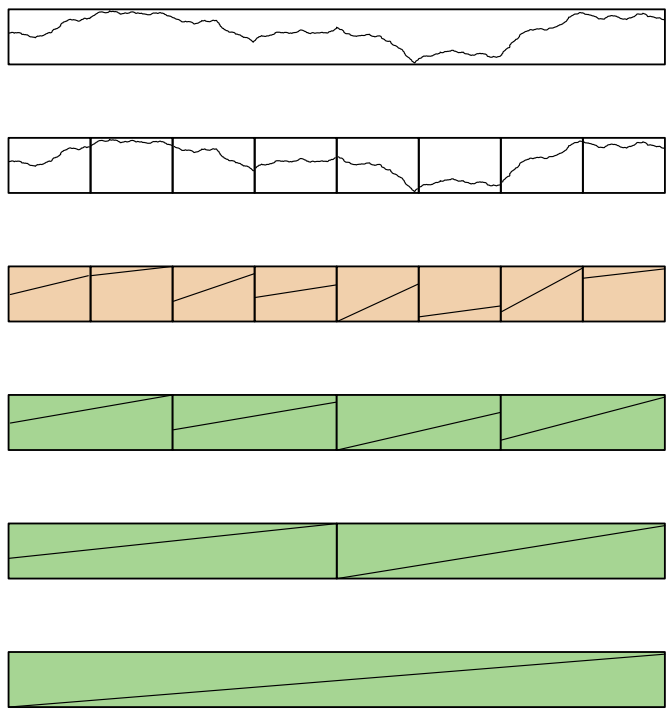


Describe Streaming Application

Describe Mergesort

- Split program into tasks
  - Partitioning
  - Presorting
  - Merging

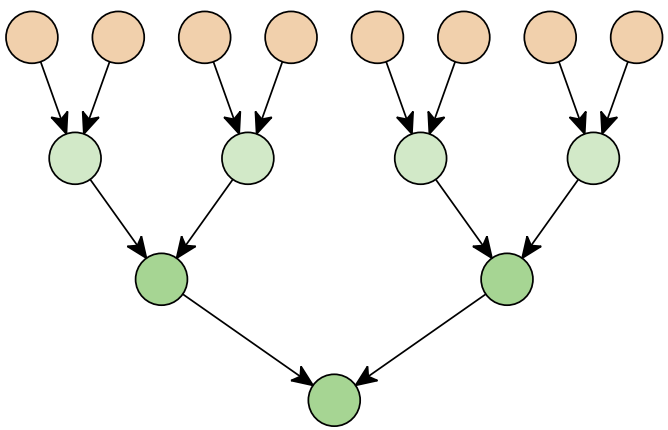


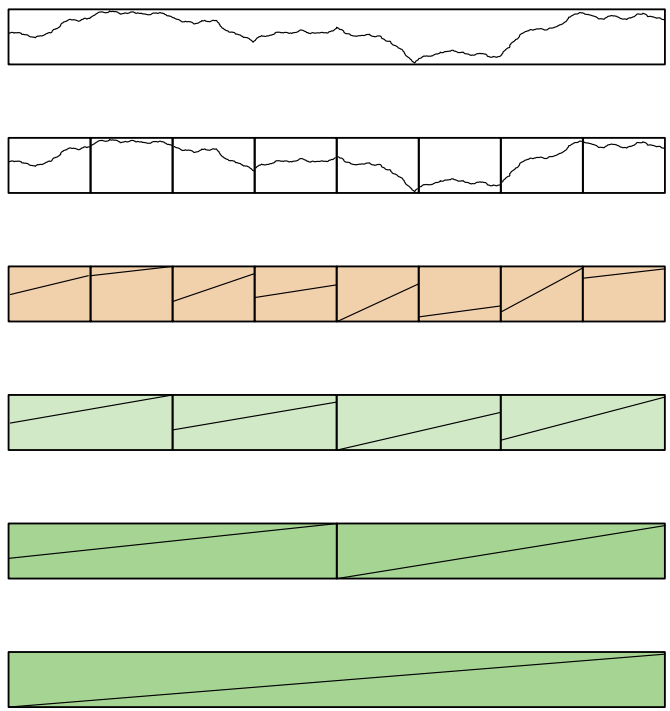


Describe Streaming Application

Describe Mergesort

- Split program into tasks
  - Partitioning
  - Presorting
  - Merging

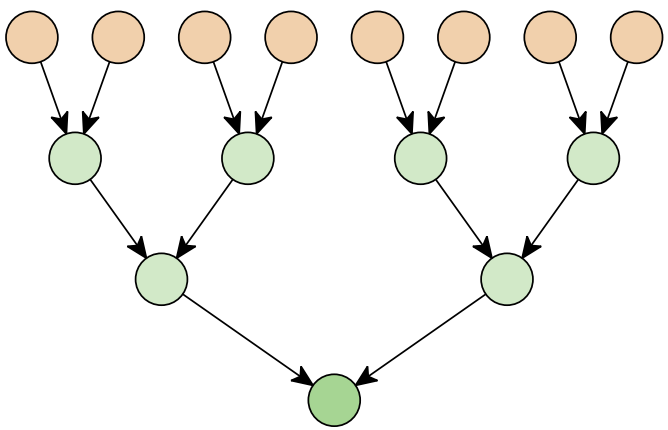


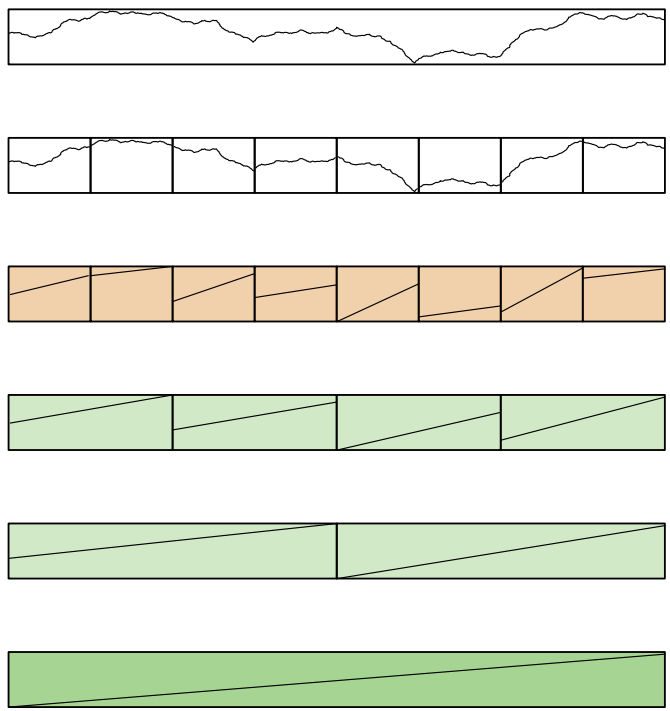


Describe Streaming Application

Describe Mergesort

- Split program into tasks
  - Partitioning
  - Presorting
  - Merging

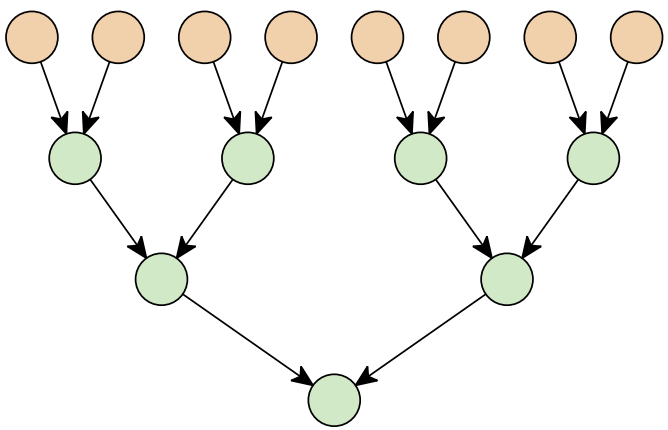




Describe Streaming Application

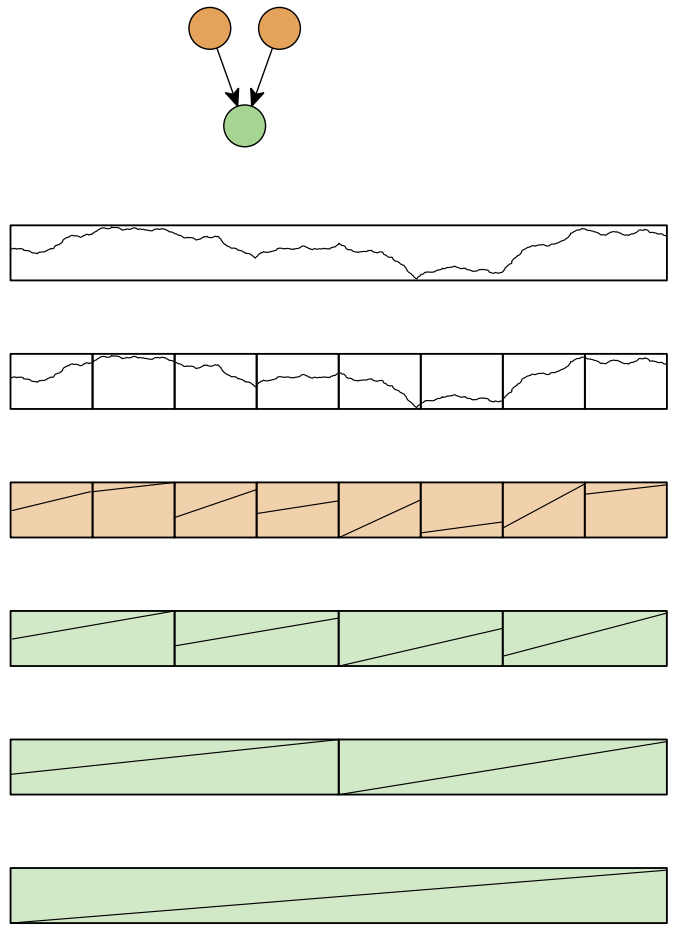
Describe Mergesort

- Split program into tasks
  - Partitioning
  - Presorting
  - Merging



Provided

- 1 level tree (*src/merge-*.graphml*)



Write Task Code

Presort already provided

- Reuses sort code of part 1 (*src/sort.cpp*)

Write Task Code

Presort already provided

- Reuses sort code of part 1 (*src/sort.cpp*)
- Implement merge task

Write Task Code

Presort already provided

- Reuses sort code of part 1 (*src/sort.cpp*)
- Implement merge task

Classic: all data available

```
1  Function merge(l, ls, r, rs, o)
2      lc ← rc ← 0;
3      while lc < ls ∧ rc < rs do
4          if l[lc] < r[rc] then
5              o[lc + rc] ← l[lc];
6              lc ← lc + 1;
7          else
8              o[lc + rc] ← r[rc];
9              rc ← rc + 1;
10         end
11     end
12     if lc = ls then
13         Copy rest of data in r to o
14     end
15     if rc = rs then
16         Copy rest of data in l to o
17     end
```

Write Task Code

Presort already provided

- Reuses sort code of part 1 (*src/sort.cpp*)
- Implement merge task

Classic: all data available

```
1  Function merge(l, ls, r, rs, o)
2      lc ← rc ← 0;
3      while lc < ls ∧ rc < rs do
4          if l[lc] < r[rc] then
5              o[lc + rc] ← l[lc];
6              lc ← lc + 1;
7          else
8              o[lc + rc] ← r[rc];
9              rc ← rc + 1;
10         end
11     end
12     if lc = ls then
13         Copy rest of data in r to o
14     end
15     if rc = rs then
16         Copy rest of data in l to o
17     end
```

Pipelined: only a portion of data at a time

- Merge items as in classic merge (lines 3 to 10)
  - But output channel may be full

Write Task Code

Presort already provided

- Reuses sort code of part 1 (*src/sort.cpp*)
- Implement merge task

Classic: all data available

```
1  Function merge(l, ls, r, rs, o)
2      lc ← rc ← 0;
3      while lc < ls ∧ rc < rs do
4          if l[lc] < r[rc] then
5              o[lc + rc] ← l[lc];
6              lc ← lc + 1;
7          else
8              o[lc + rc] ← r[rc];
9              rc ← rc + 1;
10         end
11     end
12     if lc = ls then
13         Copy rest of data in r to o
14     end
15     if rc = rs then
16         Copy rest of data in l to o
17     end
```

Pipelined: only a portion of data at a time

- Merge items as in classic merge (lines 3 to 10)
  - But output channel may be full
- When to run lines 12 to 17?
  - When corresponding channel is empty, and

Write Task Code

Presort already provided

- Reuses sort code of part 1 (*src/sort.cpp*)
- Implement merge task

Classic: all data available

```
1  Function merge(l, ls, r, rs, o)
2      lc ← rc ← 0;
3      while lc < ls ∧ rc < rs do
4          if l[lc] < r[rc] then
5              o[lc + rc] ← l[lc];
6              lc ← lc + 1;
7          else
8              o[lc + rc] ← r[rc];
9              rc ← rc + 1;
10         end
11     end
12     if lc = ls then
13         Copy rest of data in r to o
14     end
15     if rc = rs then
16         Copy rest of data in l to o
17     end
```

Pipelined: only a portion of data at a time

- Merge items as in classic merge (lines 3 to 10)
  - But output channel may be full
- When to run lines 12 to 17?
  - When corresponding channel is empty, and
  - When corresponding channel is depleted
- When is a task done?

Write Task Code

Presort already provided

- Reuses sort code of part 1 (*src/sort.cpp*)
- Implement merge task

Classic: all data available

```
1  Function merge(l, ls, r, rs, o)
2      lc ← rc ← 0;
3      while lc < ls ∧ rc < rs do
4          if l[lc] < r[rc] then
5              o[lc + rc] ← l[lc];
6              lc ← lc + 1;
7          else
8              o[lc + rc] ← r[rc];
9              rc ← rc + 1;
10         end
11     end
12     if lc = ls then
13         Copy rest of data in r to o
14     end
15     if rc = rs then
16         Copy rest of data in l to o
17     end
```

Pipelined: only a portion of data at a time

- Merge items as in classic merge (lines 3 to 10)
  - But output channel may be full
- When to run lines 12 to 17?
  - When corresponding channel is empty, and
  - When corresponding channel is depleted
- When is a task done?
  - When all channels are empty, and

Write Task Code

Presort already provided

- Reuses sort code of part 1 (*src/sort.cpp*)
- Implement merge task

Classic: all data available

```
1  Function merge(l, ls, r, rs, o)
2      lc ← rc ← 0;
3      while lc < ls ∧ rc < rs do
4          if l[lc] < r[rc] then
5              o[lc + rc] ← l[lc];
6              lc ← lc + 1;
7          else
8              o[lc + rc] ← r[rc];
9              rc ← rc + 1;
10         end
11     end
12     if lc = ls then
13         Copy rest of data in r to o
14     end
15     if rc = rs then
16         Copy rest of data in l to o
17     end
```

Pipelined: only a portion of data at a time

- Merge items as in classic merge (lines 3 to 10)
  - But output channel may be full
- When to run lines 12 to 17?
  - When corresponding channel is empty, and
  - When corresponding channel is depleted
- When is a task done?
  - When all channels are empty, and
  - When all predecessors are killed

Write Task Code

Presort already provided

- Reuses sort code of part 1 (*src/sort.cpp*)
- Implement merge task

Classic: all data available

```
1  Function merge(l, ls, r, rs, o)
2      lc ← rc ← 0;
3      while lc < ls ∧ rc < rs do
4          if l[lc] < r[rc] then
5              o[lc + rc] ← l[lc];
6              lc ← lc + 1;
7          else
8              o[lc + rc] ← r[rc];
9              rc ← rc + 1;
10         end
11     end
12     if lc = ls then
13         Copy rest of data in r to o
14     end
15     if rc = rs then
16         Copy rest of data in l to o
17     end
```

Pipelined: only a portion of data at a time

- Merge items as in classic merge (lines 3 to 10)
  - But output channel may be full
- When to run lines 12 to 17?
  - When corresponding channel is empty, and
  - When corresponding channel is depleted
- When is a task done?
  - When all channels are empty, and
  - When all predecessors are killed

See compendium and skeleton for operations on channels (peek, pop, push, empty) and tasks (killed).

Describe target platform

Platform description: *src/platform*.dat*

```
# Number of cores
param p := 1;

# Frequency levels
set F := 1200 1400 1500 1600 1700 1900 2000 2100
         2200 2300 2500 2700 2900 3000 3200 3300 3500;
param Funit := 1000000;
param Fin := 1;
set Fi[1] := 1;
```

Describe target platform

Platform description: *src/platform*.dat*

- Number of cores: *p*

```
# Number of cores
param p := 1;

# Frequency levels
set F := 1200 1400 1500 1600 1700 1900 2000 2100
         2200 2300 2500 2700 2900 3000 3200 3300 3500;
param Funit := 1000000;
param Fin := 1;
set Fi[1] := 1;
```

Describe target platform

Platform description: *src/platform*.dat*

- Number of cores: *p*
- DVFS: unused this lab

```
# Number of cores
param p := 1;

# Frequency levels
set F := 1200 1400 1500 1600 1700 1900 2000 2100
         2200 2300 2500 2700 2900 3000 3200 3300 3500;
param Funit := 1000000;
param Fin := 1;
set Fi[1] := 1;
```

Describe target platform

Platform description: *src/platform*.dat*

- Number of cores: *p*
- DVFS: unused this lab

Provided:
1 core platform description,
See below

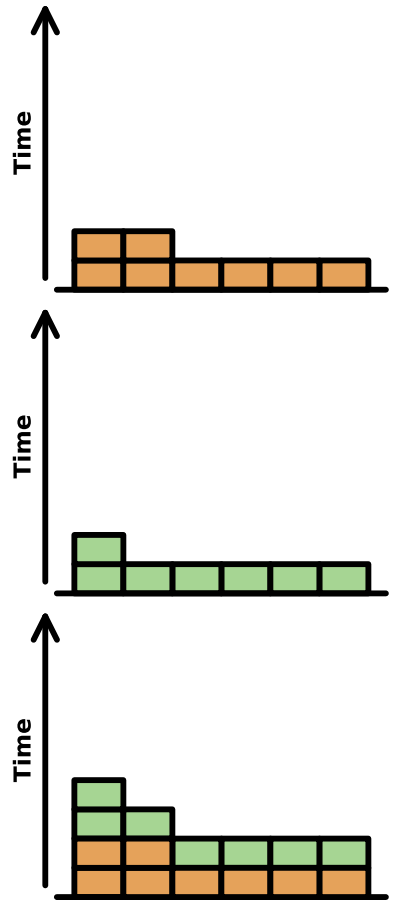
```
# Number of cores
param p := 1;

# Frequency levels
set F := 1200 1400 1500 1600 1700 1900 2000 2100
         2200 2300 2500 2700 2900 3000 3200 3300 3500;
param Funit := 1000000;
param Fin := 1;
set Fi[1] := 1;
```

Define schedule

Actual parallel speedup

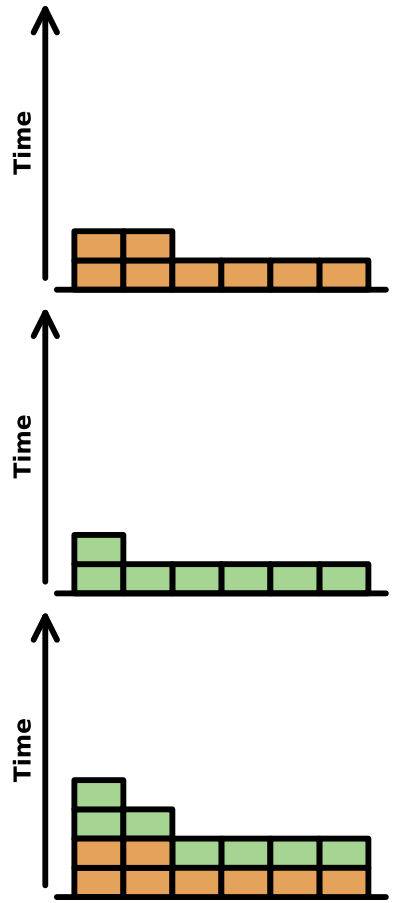
- Spread load to cores



Define schedule

Actual parallel speedup

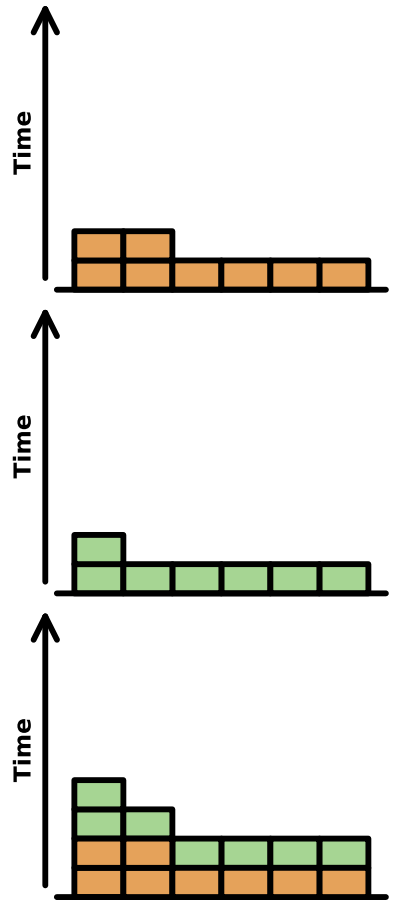
- Spread load to cores
- Streaming vs Non-Streaming tasks
  - Don’t run at the same time



Define schedule

Actual parallel speedup

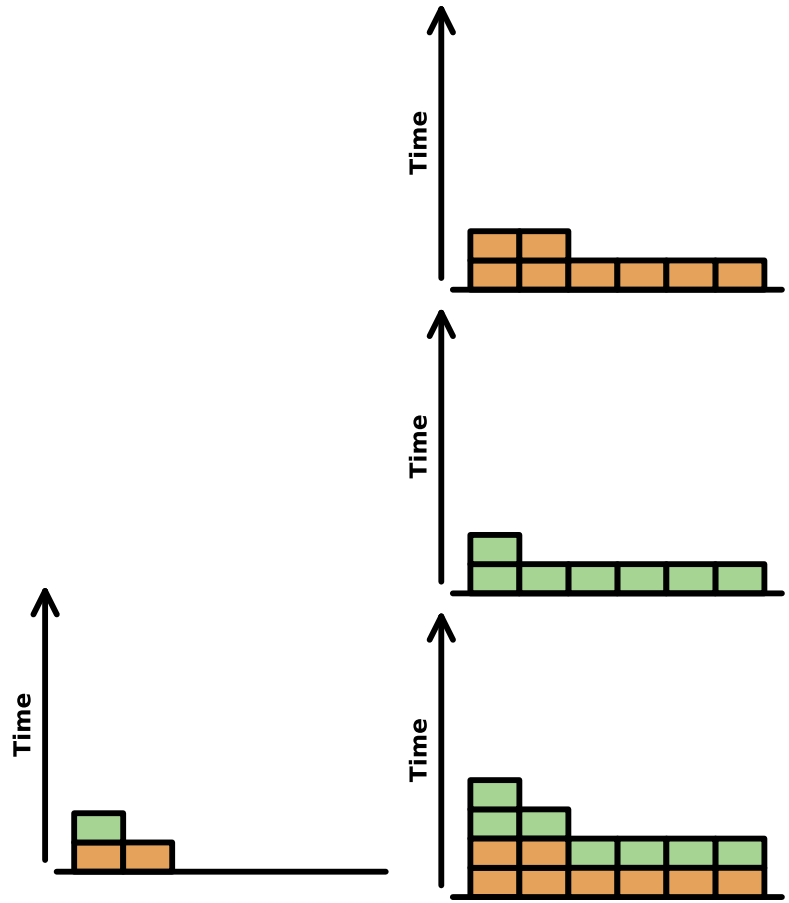
- Spread load to cores
- Streaming vs Non-Streaming tasks
  - Don't run at the same time
  - Presort: non-streaming
  - Merge: streaming



Define schedule

Actual parallel speedup

- Spread load to cores
- Streaming vs Non-Streaming tasks
  - Don't run at the same time
  - Presort: non-streaming
  - Merge: streaming



Example:

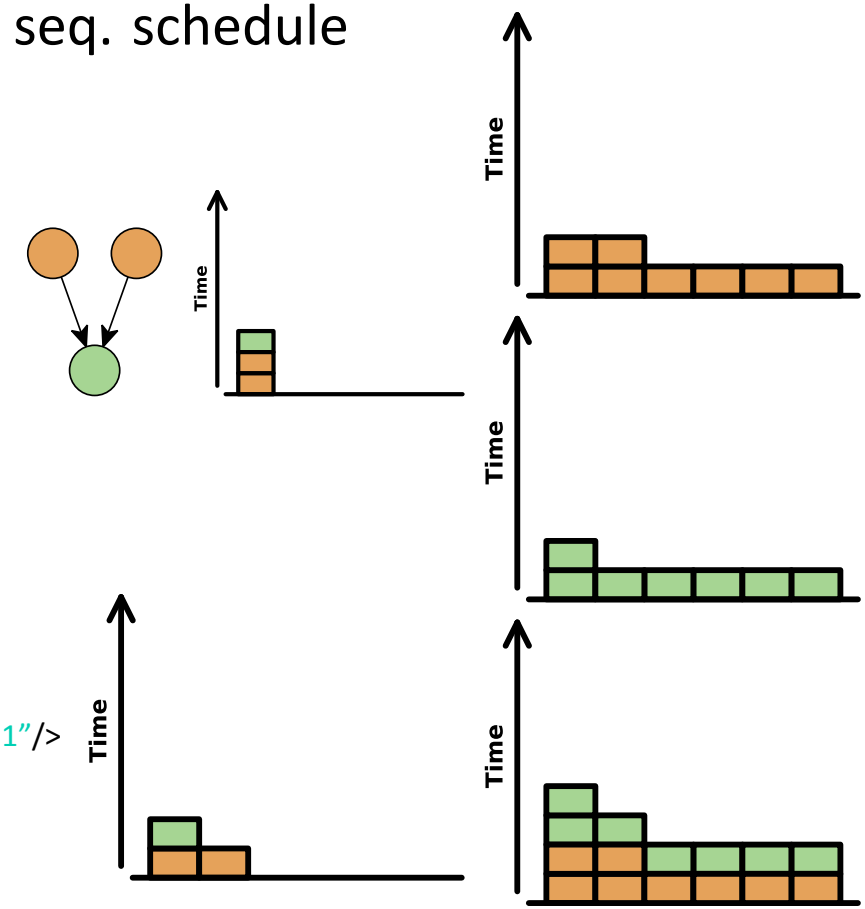
```
<?xml version="1.0" encoding="UTF-8"?>
<schedule name="nicme26" appname="pipelined_merge">
 <core coreid="1">
  <task name="leaf_1" start="0" frequency="3500" width="1" workload="1"/>
  <task name="root" start="2.85e-7" frequency="3500" width="1" workload="1"/>
 </core>
 <core coreid="2">
  <task name="leaf_2" start="0" frequency="3500" width="1" workload="1"/>
 </core>
</schedule>
```

Define schedule

Actual parallel speedup

- Spread load to cores
- Streaming vs Non-Streaming tasks
  - Don't run at the same time
  - Presort: non-streaming
  - Merge: streaming

Provided: 1-level seq. schedule



Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<schedule name="nicme26" appname="pipelined_merge">
 <core coreid="1">
  <task name="leaf_1" start="0" frequency="3500" width="1" workload="1"/>
  <task name="root" start="2.85e-7" frequency="3500" width="1" workload="1"/>
 </core>
 <core coreid="2">
  <task name="leaf_2" start="0" frequency="3500" width="1" workload="1"/>
 </core>
</schedule>
```

Compile & Test Application

Compile for debugging

- Task graph: *src/merge.graphml*
- Platform description: *src/platform.dat*
- Schedule: generated by LPT in *src/merge.schedule.xml*

Compile & Test Application

Compile for debugging

- Task graph: *src/merge.graphml*
- Platform description: *src/platform.dat*
- Schedule: generated by LPT in *src/merge.schedule.xml*
- Compile: *make CONFIG=DEBUG NB_THREADS=x*

Compile & Test Application

Compile for debugging

- Task graph: *src/merge.graphml*
- Platform description: *src/platform.dat*
- Schedule: generated by LPT in *src/merge.schedule.xml*
- Compile: *make CONFIG=DEBUG NB_THREADS=x*

Testing

- Generate inputs with *src/generate*
- Use *dump* to inspect input data file

Compile & Test Application

Compile for debugging

- Task graph: *src/merge.graphml*
- Platform description: *src/platform.dat*
- Schedule: generated by LPT in *src/merge.schedule.xml*
- Compile: *make CONFIG=DEBUG NB_THREADS=x*

Testing

- Generate inputs with *src/generate*
- Use *dump* to inspect input data file
- Test correctness of *src/drake-sort* and *src/parallel-sort*
- Use *backtrace* from crashed programs' outputs to locate errors

Measure Performance

Skeleton compiles variants for 1 to 6 cores

- Handwritten parallel sort: *NB_THREADS* preprocessor symbol in *src/sort.cpp*.

Measure Performance

Skeleton compiles variants for 1 to 6 cores

- Handwritten parallel sort: *NB_THREADS* preprocessor symbol in *src/sort.cpp*.
- Drake-based parallel sort for *x* cores
  - Task graph: *src/merge-x.graphml*
  - Platform description: *src/merge-x.graphml*
  - Schedule: *src/schedule-x.xml*

Measure Performance

Skeleton compiles variants for 1 to 6 cores

- Handwritten parallel sort: *NB_THREADS* preprocessor symbol in *src/sort.cpp*.
- Drake-based parallel sort for *x* cores
  - Task graph: *src/merge-x.graphml*
  - Platform description: *src/merge-x.graphml*
  - Schedule: *src/schedule-x.xml*

Begin process:

- Compilation: *freja compile --log compile.log*
  - Takes a while with deep merge trees
  - See compilation details in *compile.log*

Measure Performance

Skeleton compiles variants for 1 to 6 cores

- Handwritten parallel sort: *NB_THREADS* preprocessor symbol in *src/sort.cpp*.
- Drake-based parallel sort for *x* cores
  - Task graph: *src/merge-x.graphml*
  - Platform description: *src/merge-x.graphml*
  - Schedule: *src/schedule-x.xml*

Begin process:

- Compilation: *freja compile --log compile.log*
  - Takes a while with deep merge trees
  - See compilation details in *compile.log*
- Experiment: *freja run <name> --log <name>.log*
  - Remove directory <name> before running again

Measure Performance

Skeleton compiles variants for 1 to 6 cores

- Handwritten parallel sort: *NB_THREADS* preprocessor symbol in *src/sort.cpp*.
- Drake-based parallel sort for *x* cores
  - Task graph: *src/merge-x.graphml*
  - Platform description: *src/merge-x.graphml*
  - Schedule: *src/schedule-x.xml*

Begin process:

- Compilation: *freja compile --log compile.log*
  - Takes a while with deep merge trees
  - See compilation details in *compile.log*
- Experiment: *freja run <name> --log <name>.log*
  - Remove directory <name> before running again
- Plotting: *./drawplots*

Lab 3: Inspiration

- Quick sort: `http://www.youtube.com/watch?v=ywWBy6J5gz8`
- Merge sort: `http://www.youtube.com/watch?v=XaqR3G_NVoo`
- Select sort: `http://www.youtube.com/watch?v=Ns4TPTC8whw`
- Bubble sort: `http://www.youtube.com/watch?v=lyZQPjUT5B4`
- Shell sort: `http://www.youtube.com/watch?v=CmPA7zE8mx0`
- Insert sort: `http://www.youtube.com/watch?v=ROalU37913U`
- Bogo sort: `http://en.wikipedia.org/wiki/Bogosort`
- Ineffective sorting: `http://xkcd.com/1185/`

Lab 3: Anti-inspiration

Figure: Nobody will pass these algorithms, even parallelized. Creative Commons Attribution-NonCommercial 2.5 License http://xkcd.com/1185/

Optional: High performance parallel programming challenge 2015

- Design and implement the fastest algorithms
- One round for CPUs and another on GPUS
- Participation of both rounds and on time submission is mandatory to win the prize
- Participation to the challenge at all is optional
- Prize: cinema tickets and certificate

Suggested planning

- Week 45: Lecture – Shared memory and non-blocking synchronization
- Week 46: Lab 1 – Load balancing
  - Soft deadline: Friday 11/13
- Week 46-47: Lecture – Parallel sorting algorithms
- Week 47: Lab 2 – Non-blocking synchronization
  - Soft deadline: Friday 11/20
- Week 48: Lab 3 – Parallel sorting
  - Soft deadline: Friday 11/27
- Thursday 12/18: Deadline for the CPU and GPU parallel sorting challenge
- Friday 12/19: Challenge prize ceremony and hard deadline for labs

Final remarks

- **Do your homework before the lab session**
- KISS: Keep It Simple and Stupid
  - Donald Knuth: “Premature optimization is the root of all evil”
  - Simple code often fast and good base for later optimizations
  - Start with the most possibly simple sequential code until it works
  - Then, parallelize it the most simple way you can think of, discard optimizations for now
  - Only then, improve your code: running in-place, data locality issues, thread pools, etc.
  - Read `http://en.wikipedia.org/wiki/Program_optimization`, section ”when to optimize”
- Modify lab skeletons at will!
  The exercice consists in the demonstration of understanding, not just the implementation of an expected outcome.

Final remarks

- Tight deadline
  - Implement your lab before the lab sessions
  - Use the lab session to run measurements and demonstrate your work
  - Contact your lab assistant to get help
- Help each other: find inspiration with discussions between groups
  - But code and explain solutions yourself
- **Do your homework before the lab session**
- Find me in my office 3B:488
- Introductory practice exercises on C, pthreads and measurements
  - `http://www.ida.liu.se/~nicme26/tddd56.en.shtml`, labs 0a, 0b
  - `http://www.ida.liu.se/~nicme26/measuring.en.shtml`
- Google is your friend (or claims to be so)
- A lot of useful information in the (long) lab compendium. Read them!

End of lesson 1

Next lesson: theory exercises

Thank you for your attention