Quantifying Energy Consumption for Practical Fork-Join Parallelism on an Embedded Real-Time Operating System

Antonio Paolillo - Ph.D student & software engineer

24th ACM International Conference on Real-Time Networks and Systems

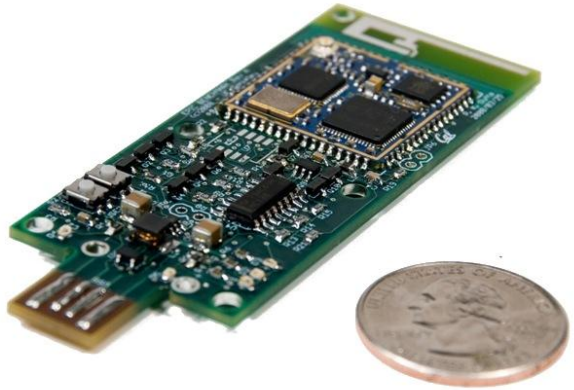
21th October 2016

Goal:

Goal:

Parallelism helps to **reduce energy** while meeting **real-time requirements**

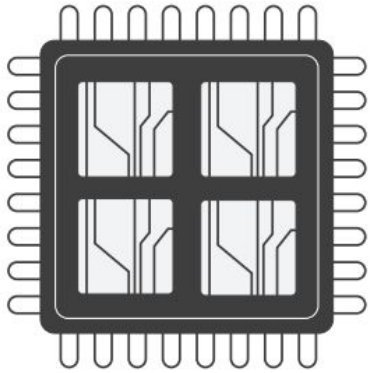
Quantifying Energy Consumption for Practical Fork-Join Parallelism on an Embedded Real-Time Operating System



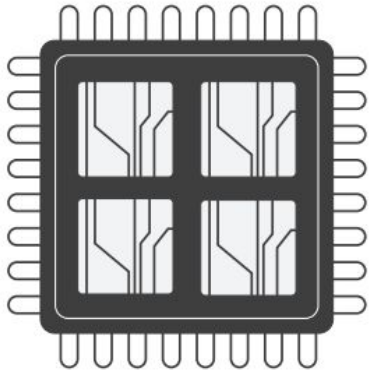
Quantifying Energy Consumption for Practical Fork-Join Parallelism on an **Embedded** Real-Time Operating System

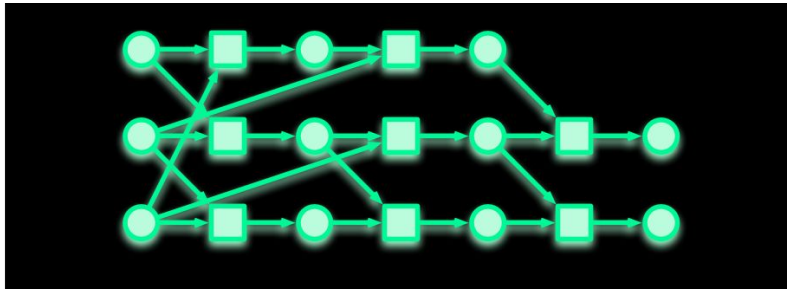
Quantifying **Energy Consumption** for Practical Fork-Join Parallelism on an Embedded Real-Time Operating System

Quantifying Energy Consumption for Practical Fork-Join Parallelism on an Embedded **Real-Time** Operating System

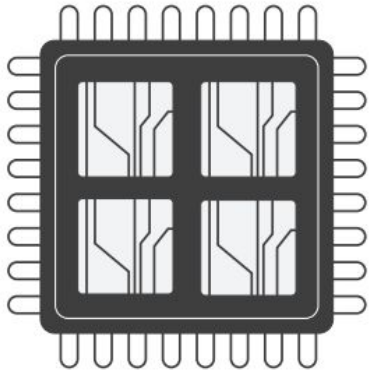


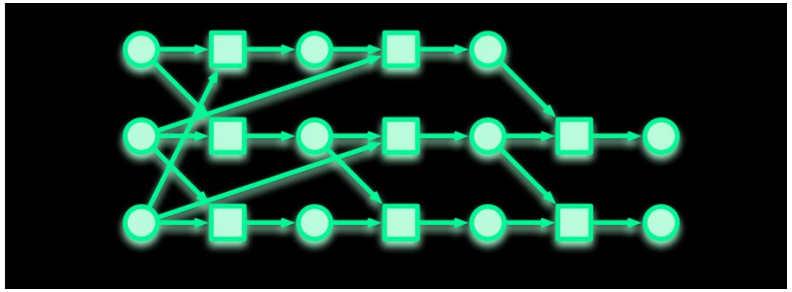
Quantifying Energy Consumption for Practical Fork-Join **Parallelism** on an Embedded Real-Time Operating System





Quantifying Energy Consumption for Practical **Fork-Join Parallelism** on an Embedded Real-Time Operating System

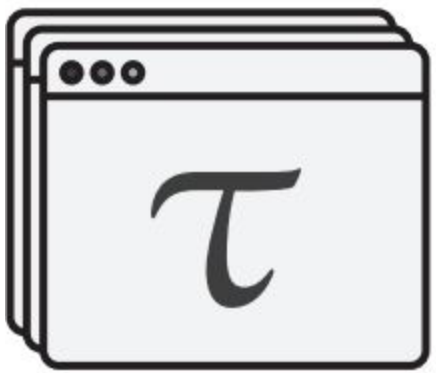


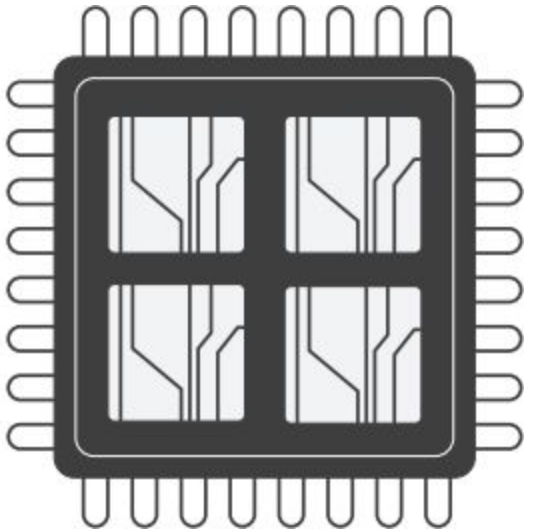


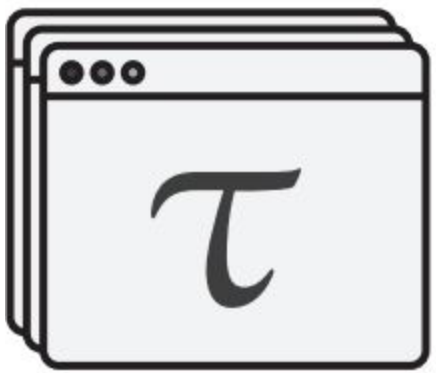
OpenMP™

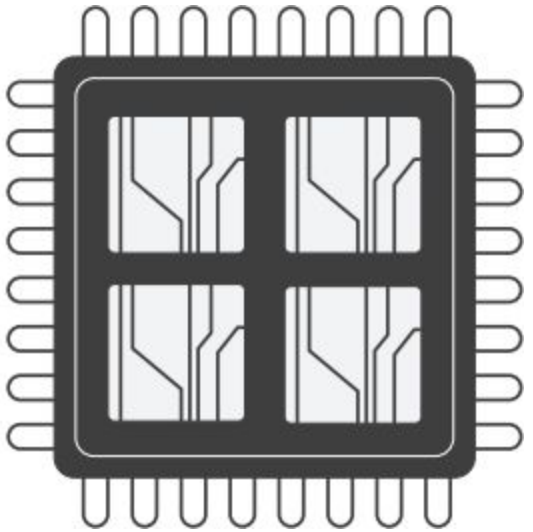
Quantifying Energy Consumption for **Practical Fork-Join Parallelism** on an Embedded Real-Time Operating System

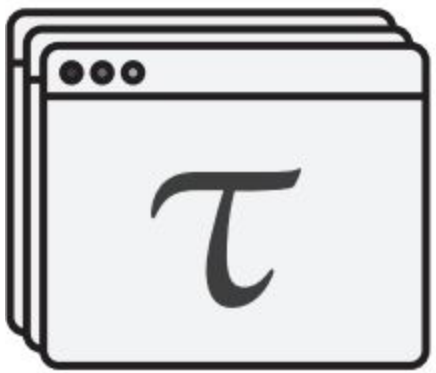
Quantifying Energy Consumption for Practical Fork-Join Parallelism on an Embedded **Real-Time Operating System**

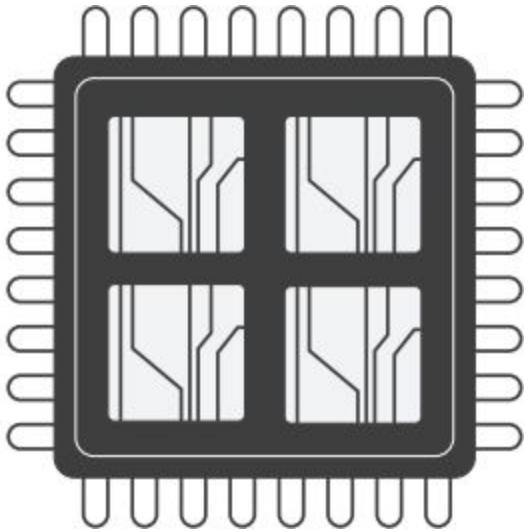

τ

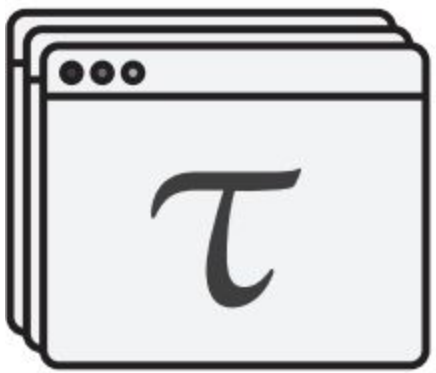


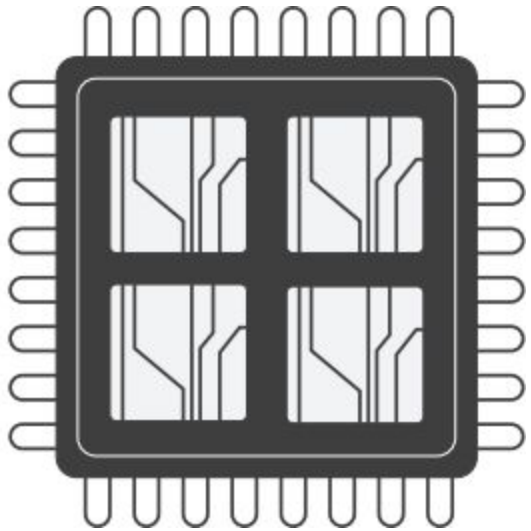

τ

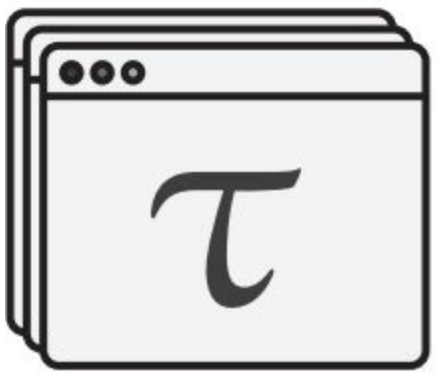


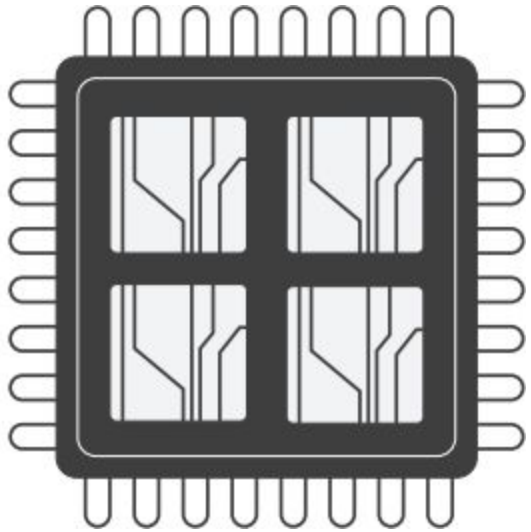

τ

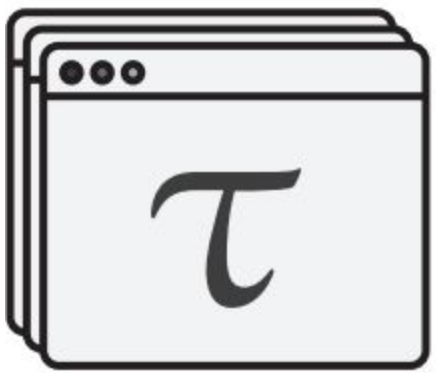


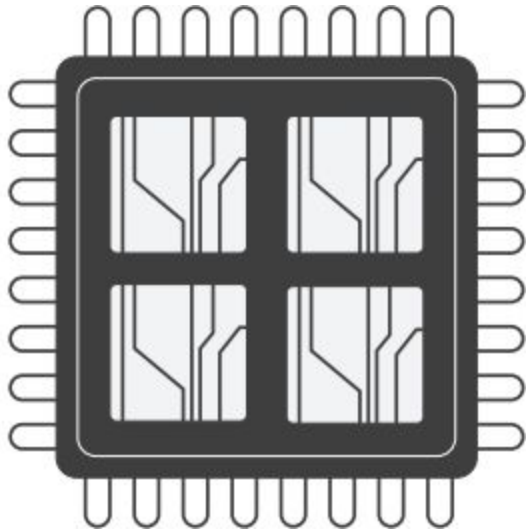

τ

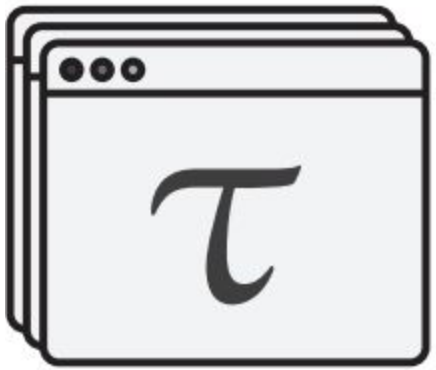


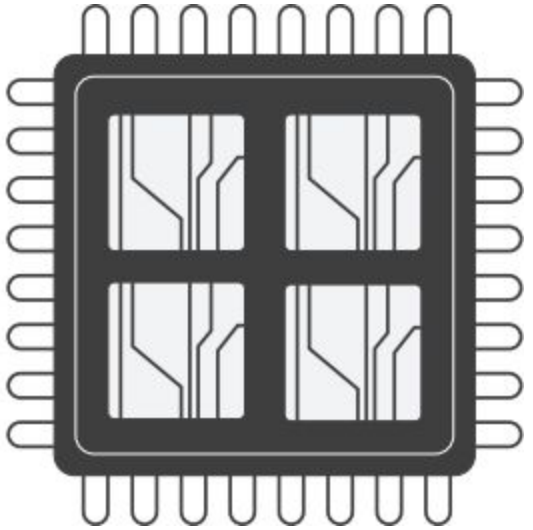

τ

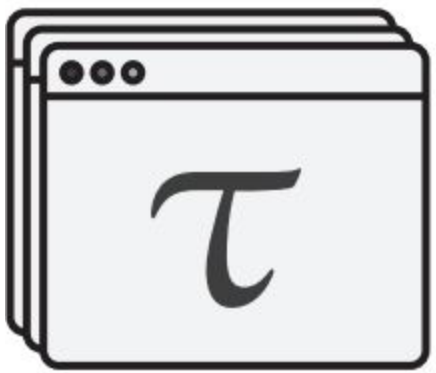


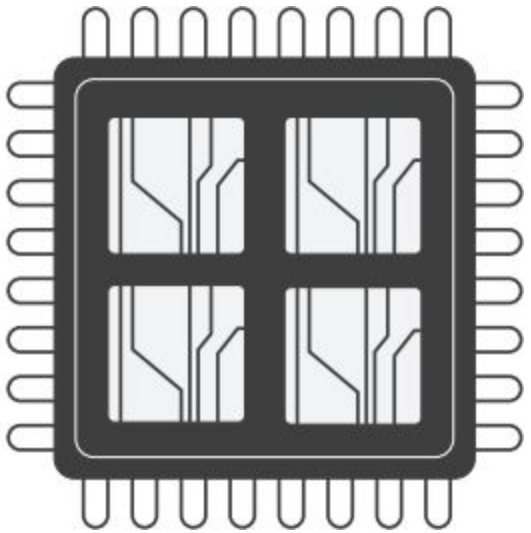

τ

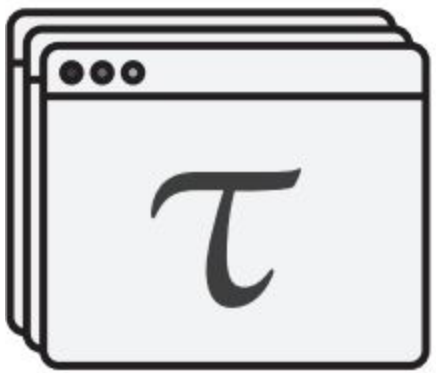


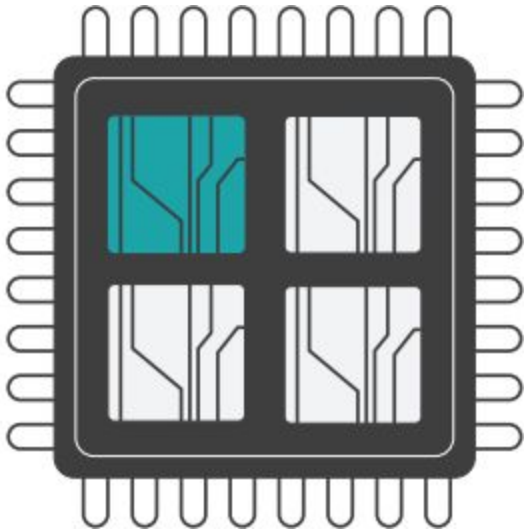

τ

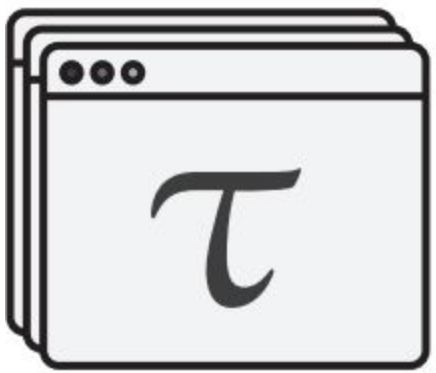



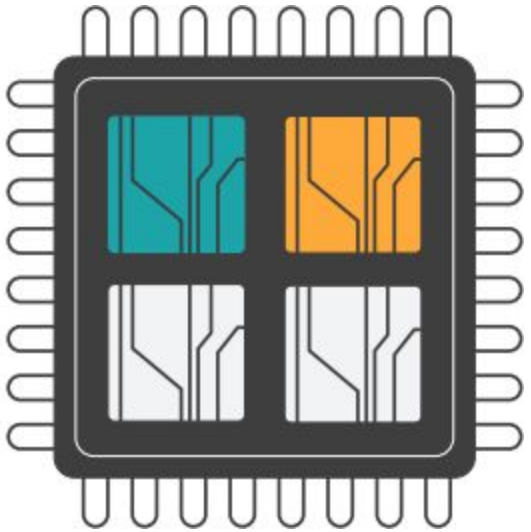

τ

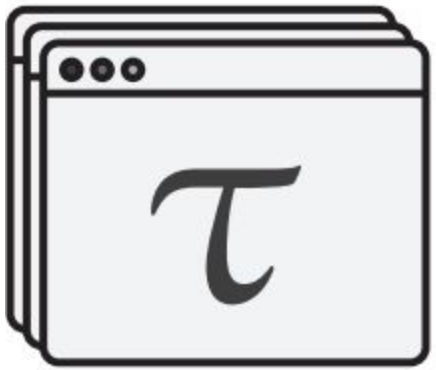



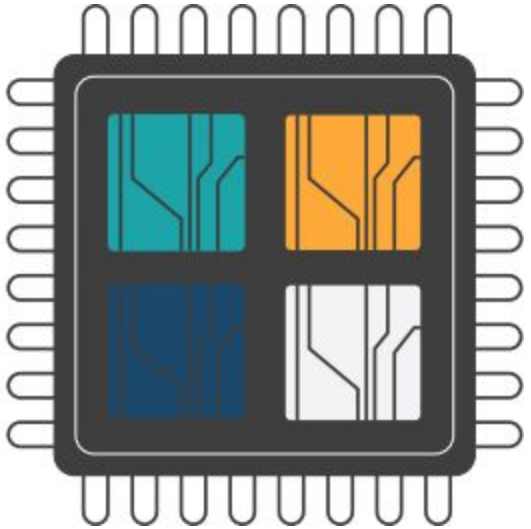

τ

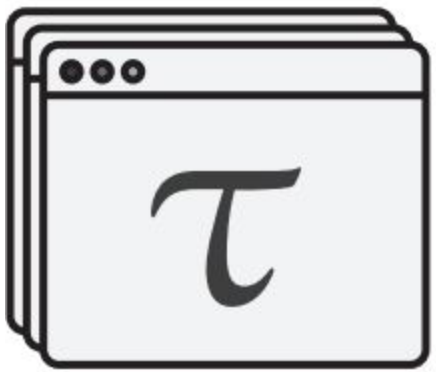



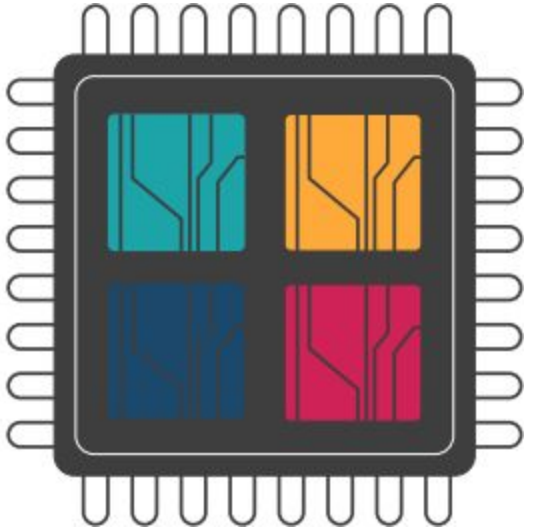

τ

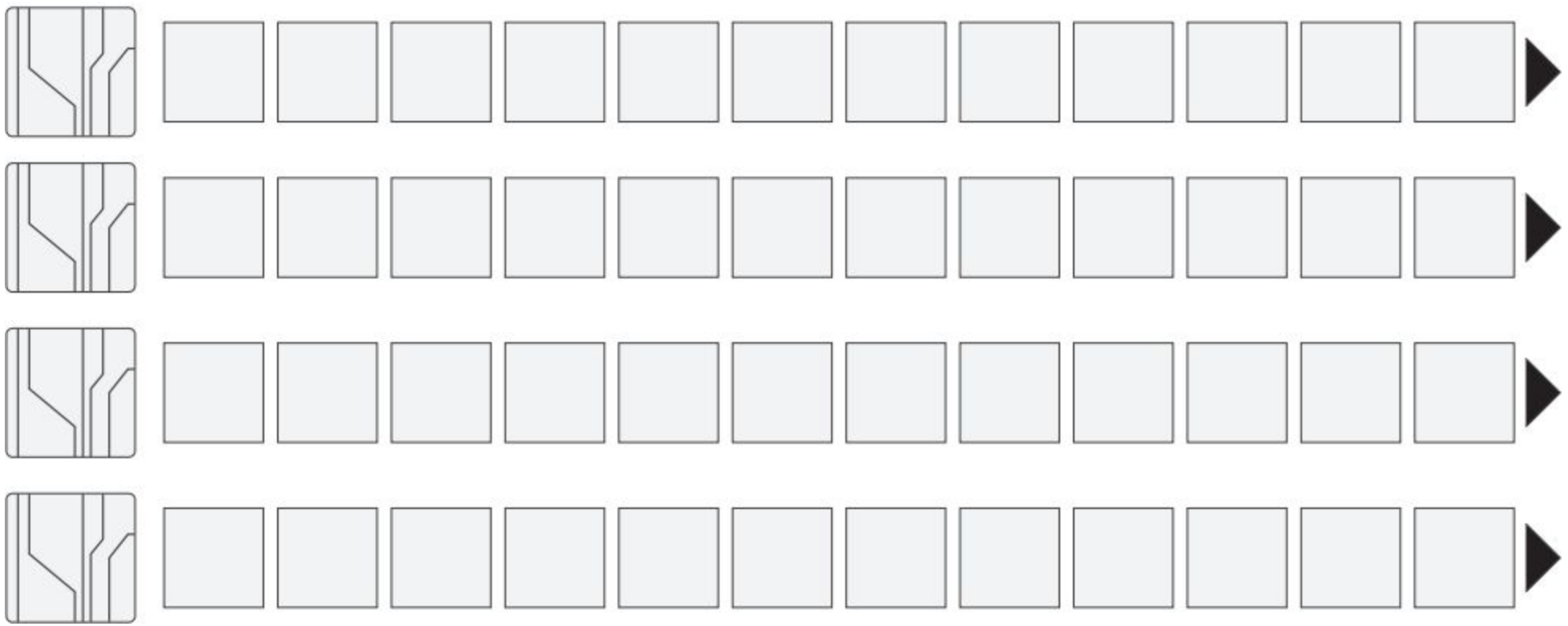

τ

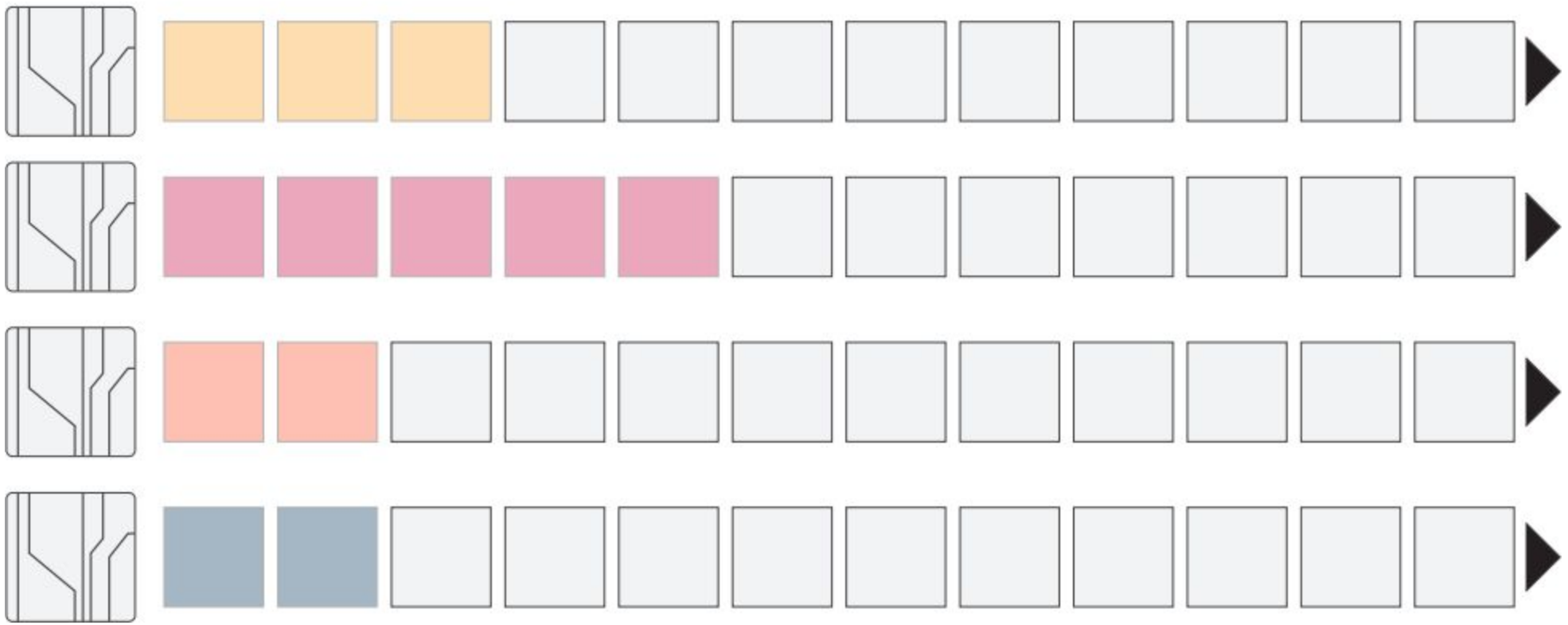

τ

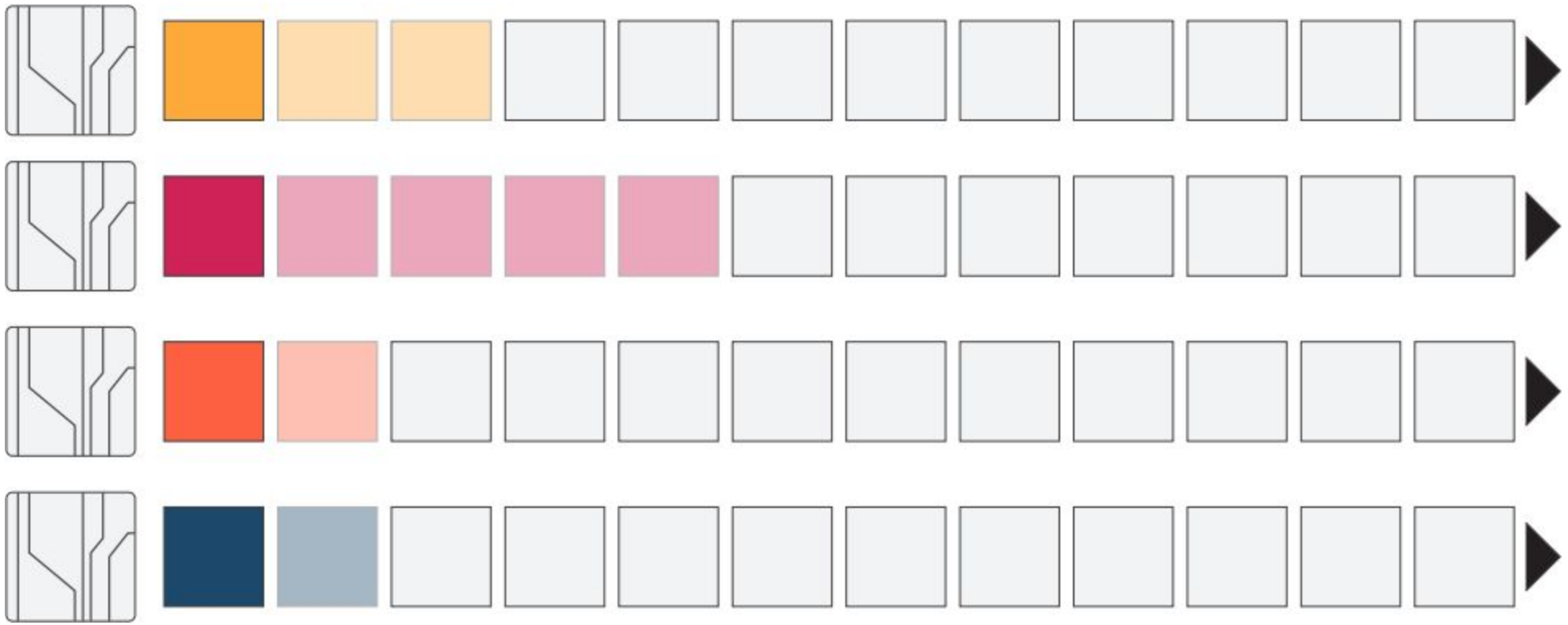
Sequential schedule



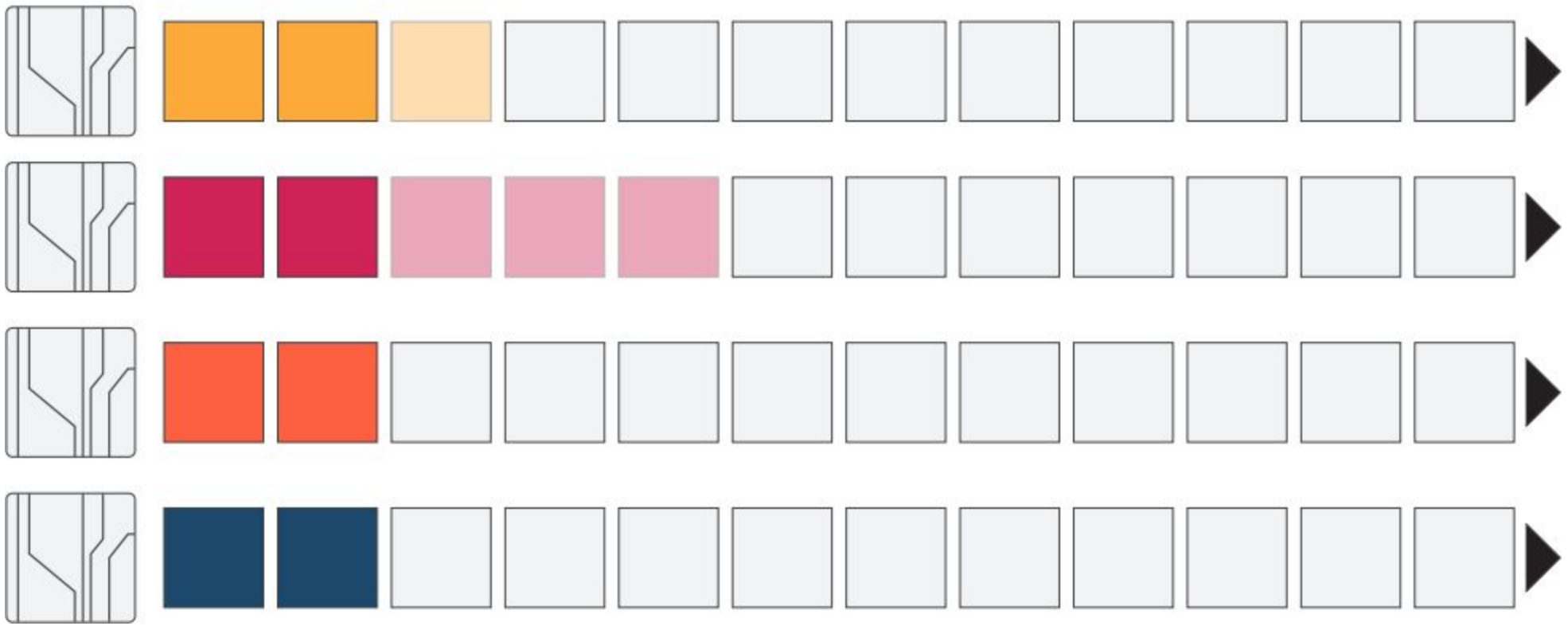
Sequential schedule



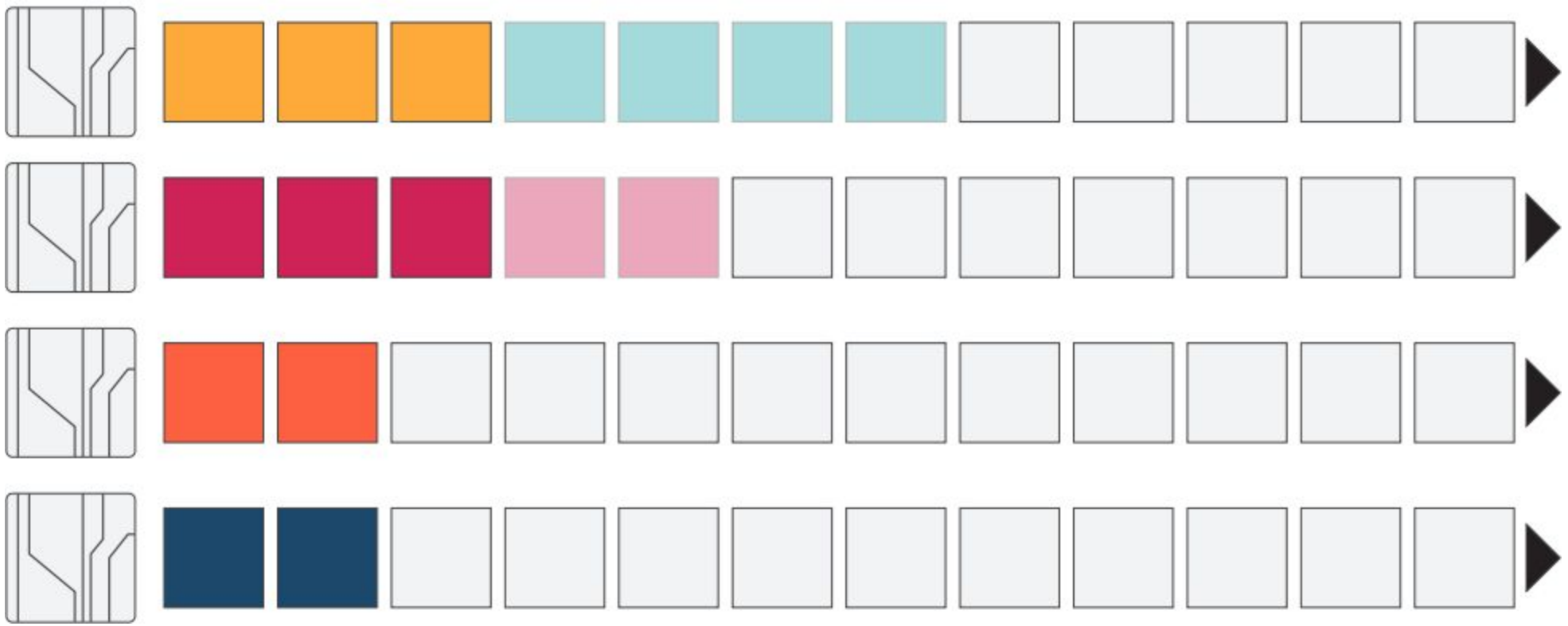
Sequential schedule



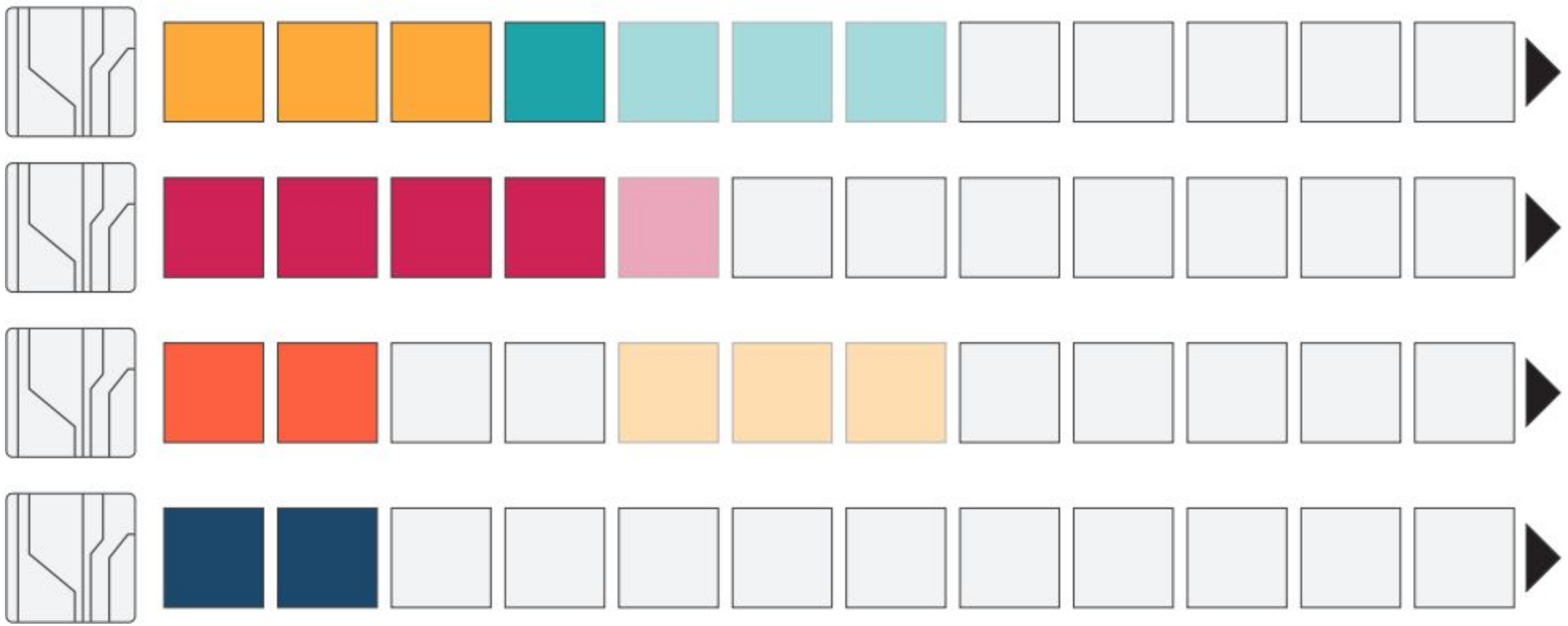
Sequential schedule



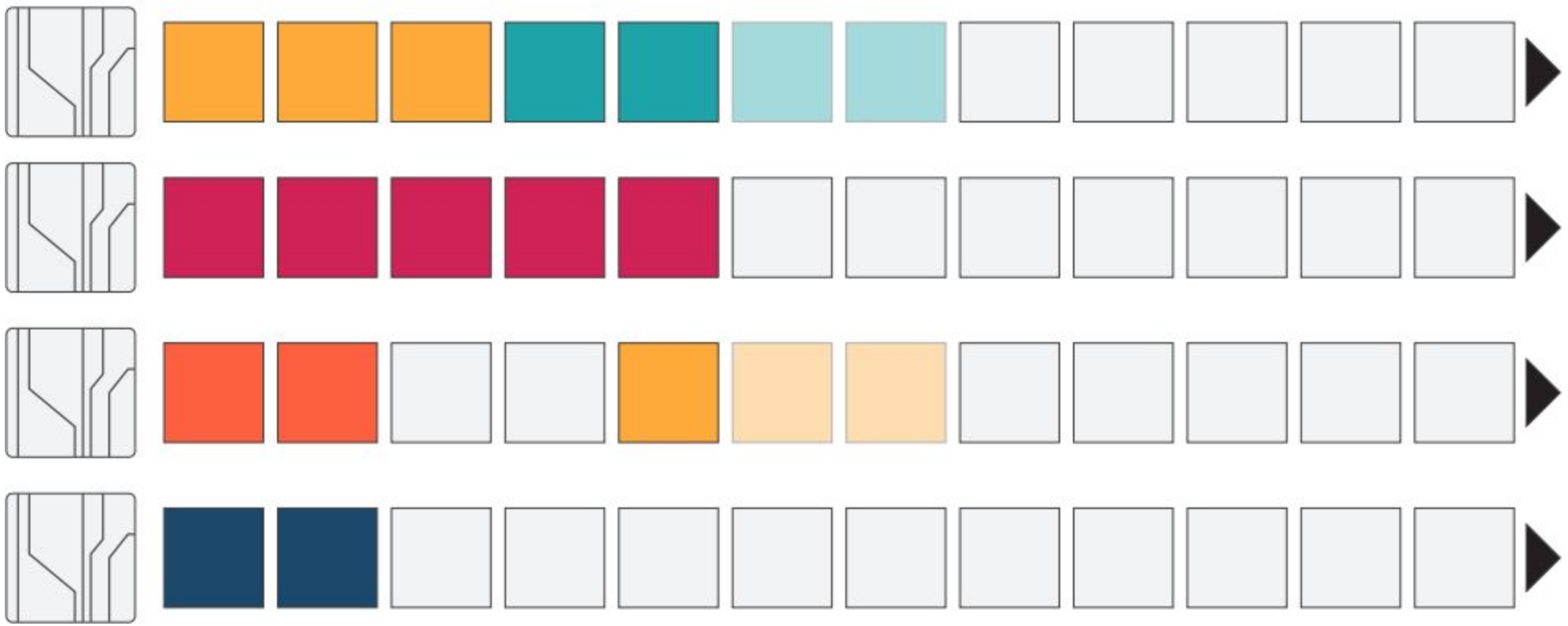
Sequential schedule



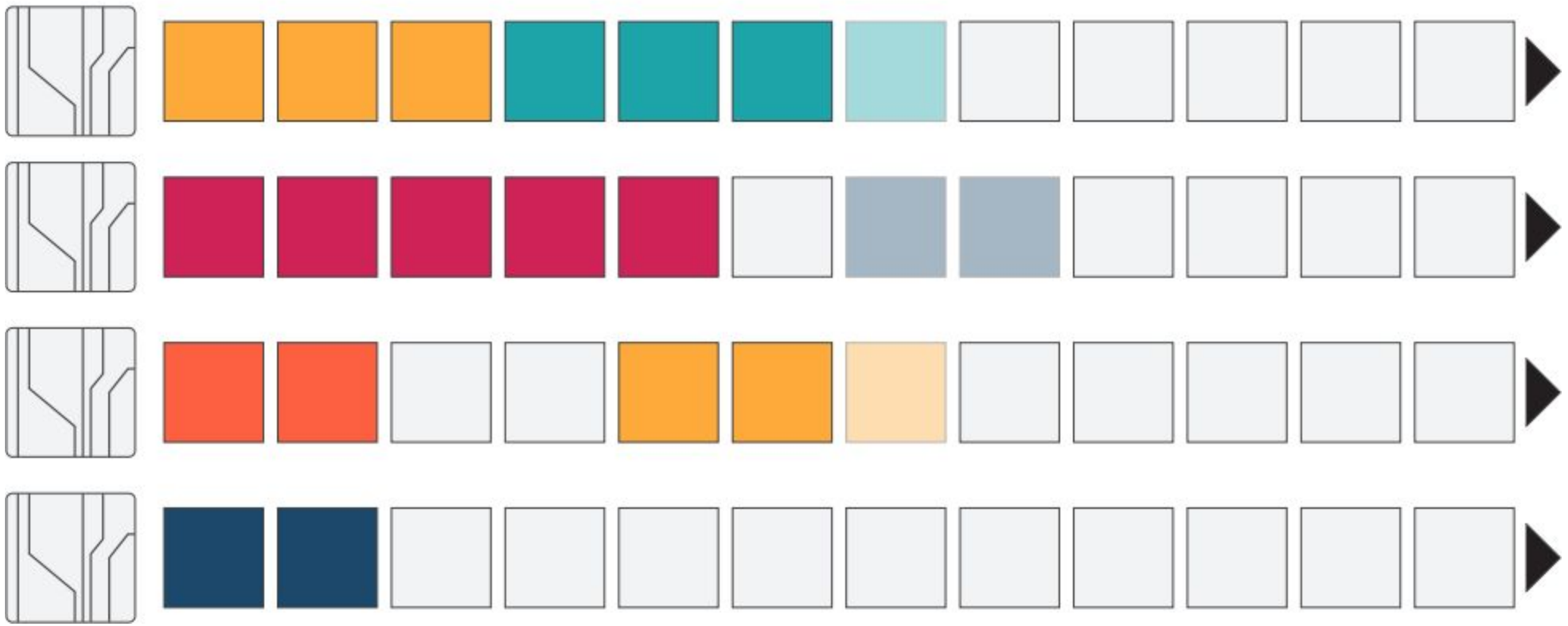
Sequential schedule



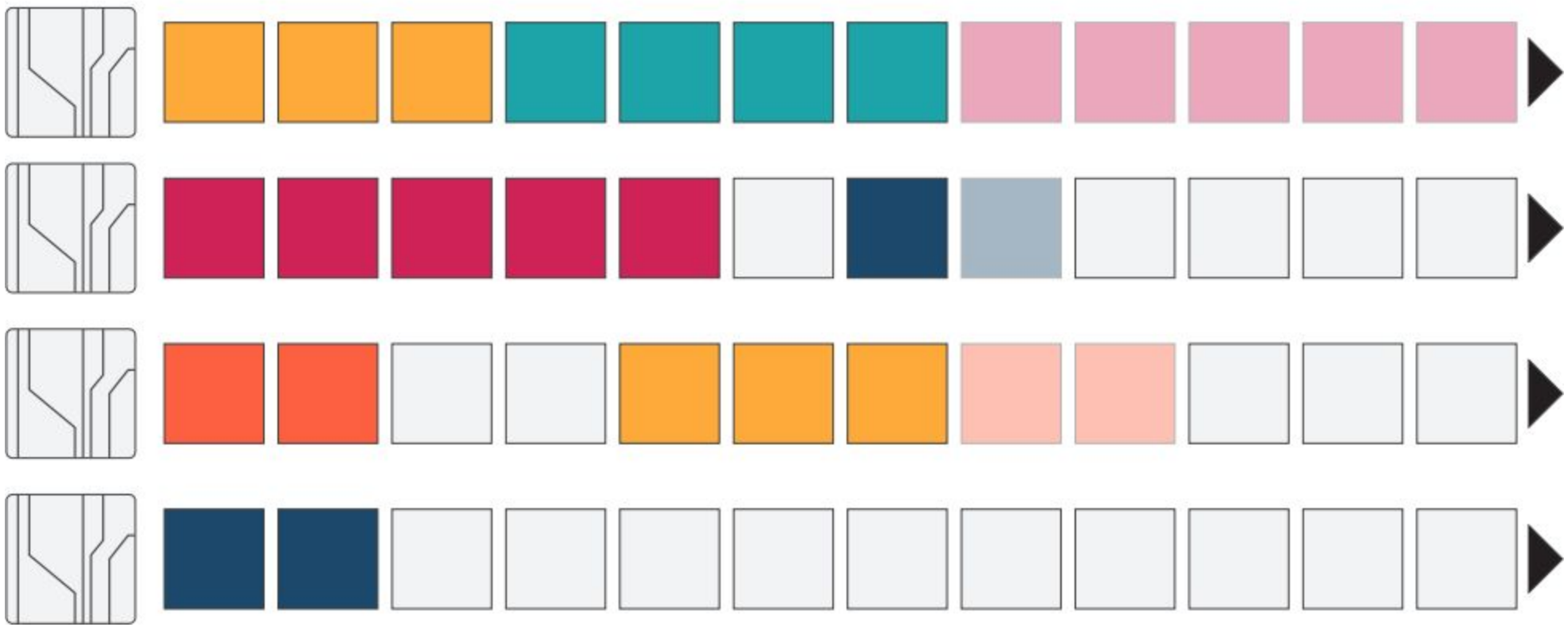
Sequential schedule



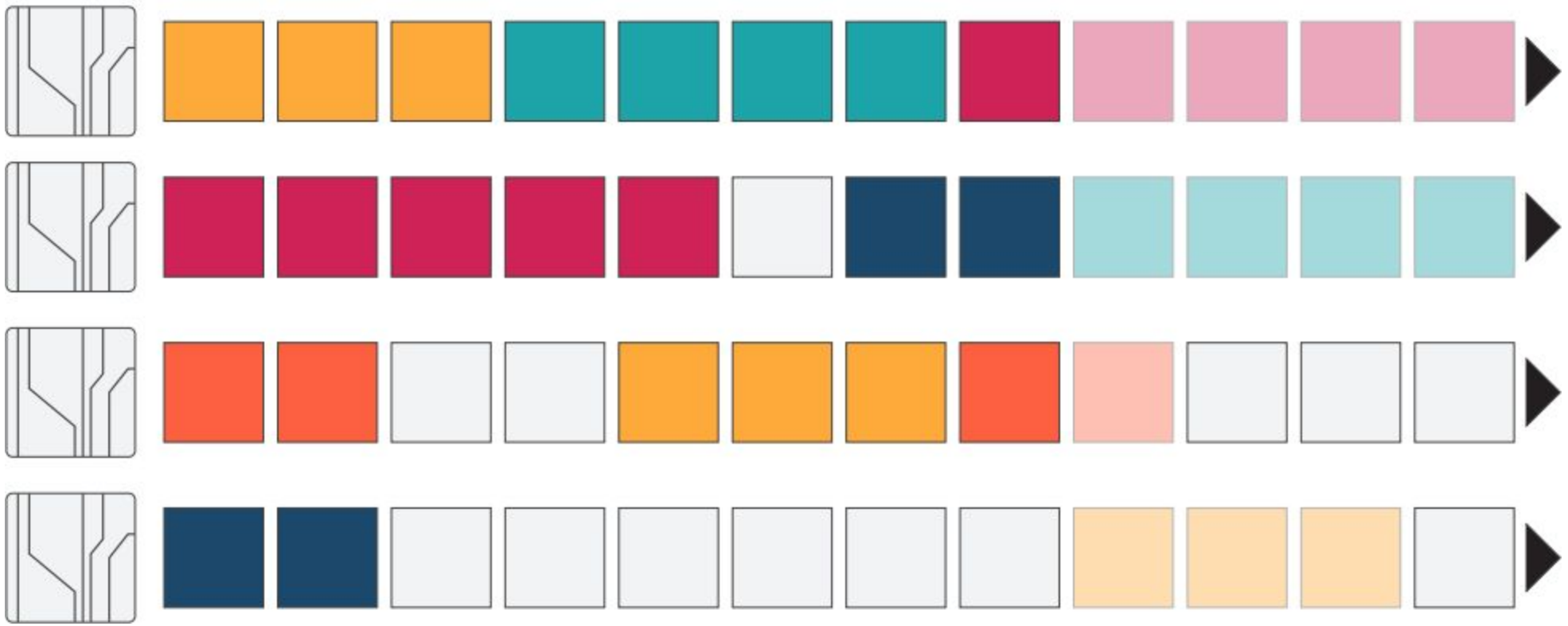
Sequential schedule



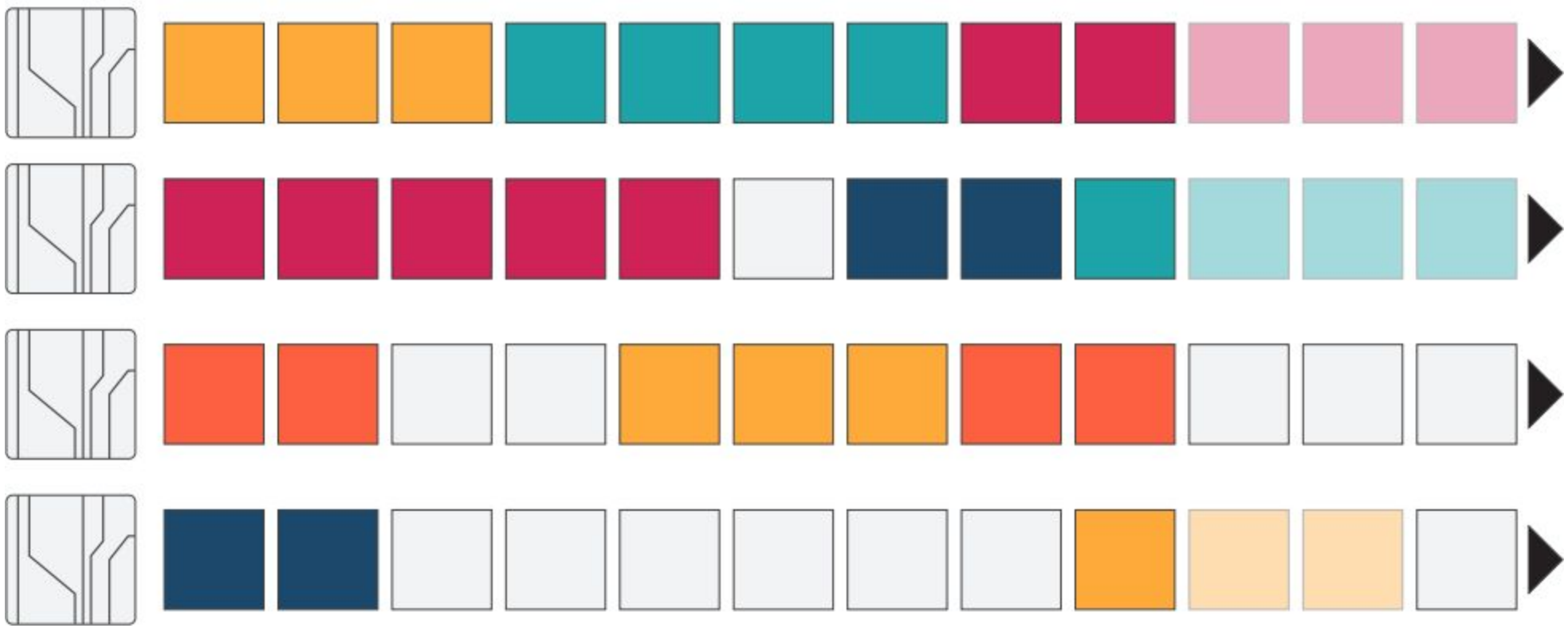
Sequential schedule



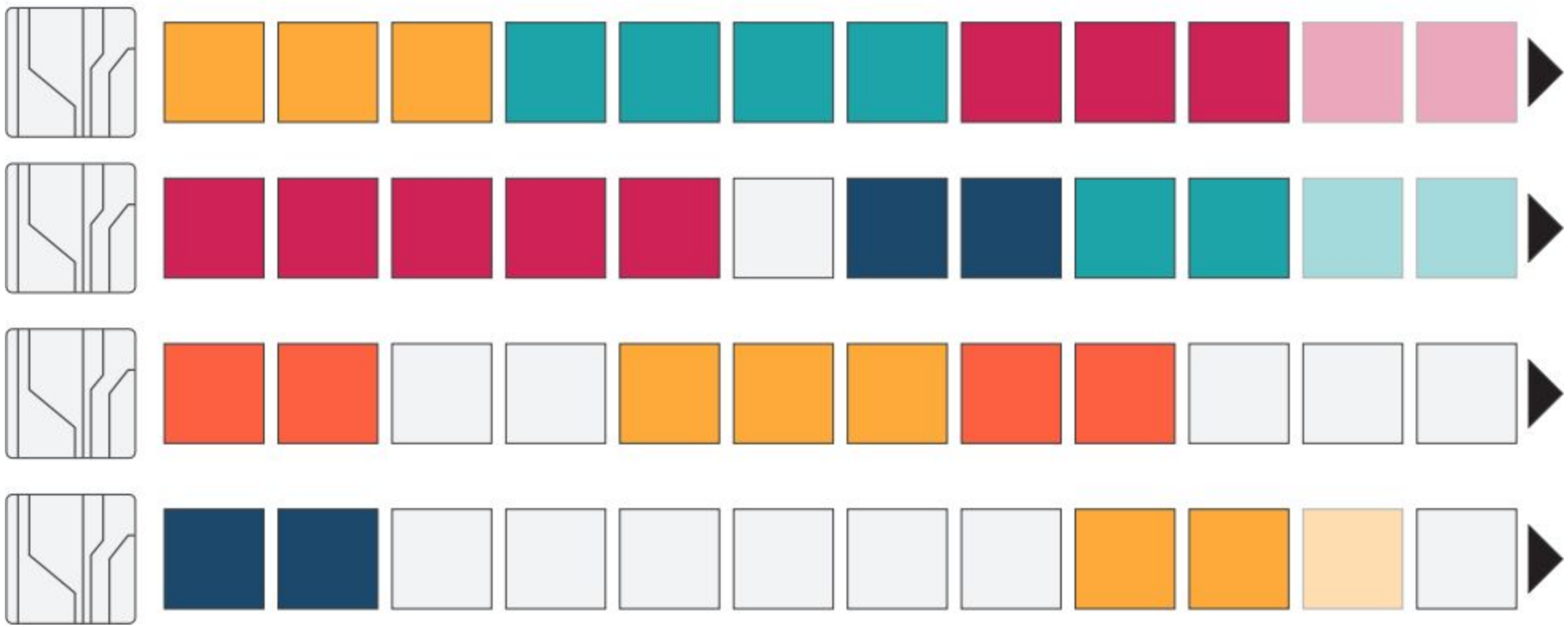
Sequential schedule



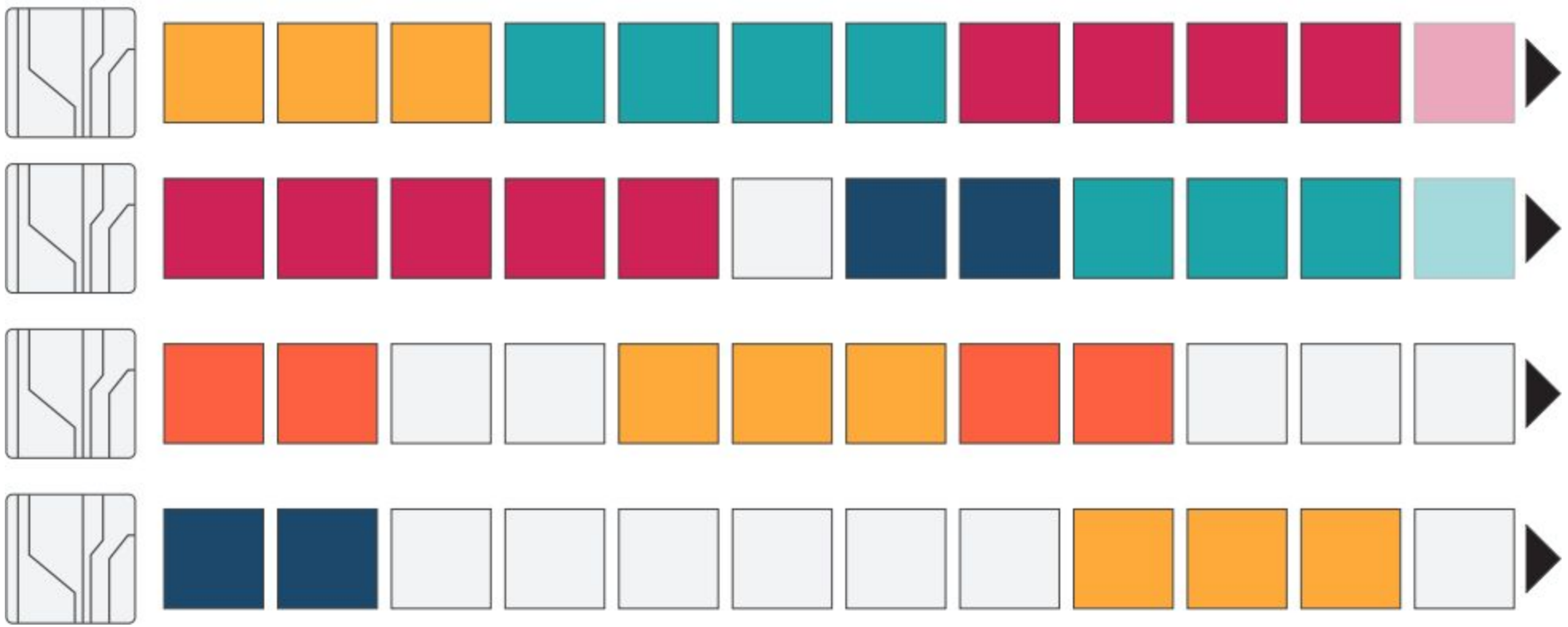
Sequential schedule



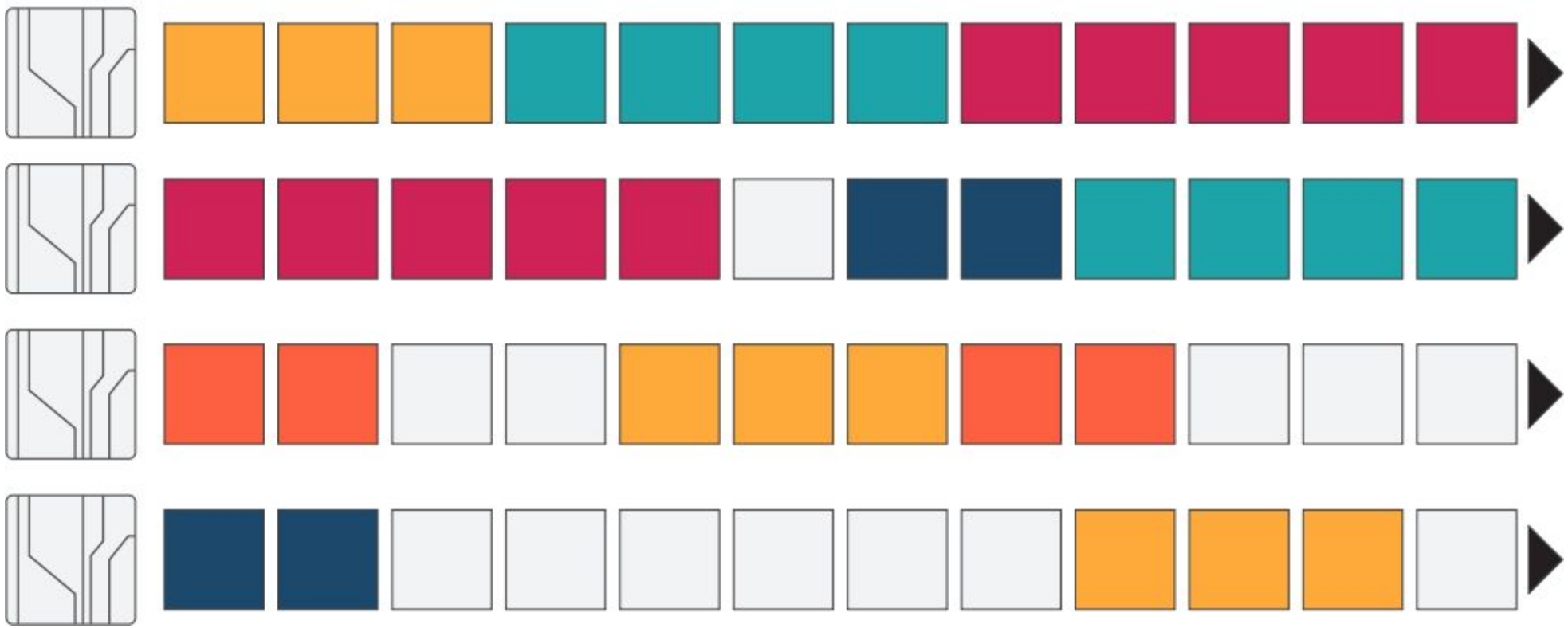
Sequential schedule



Sequential schedule



Sequential schedule

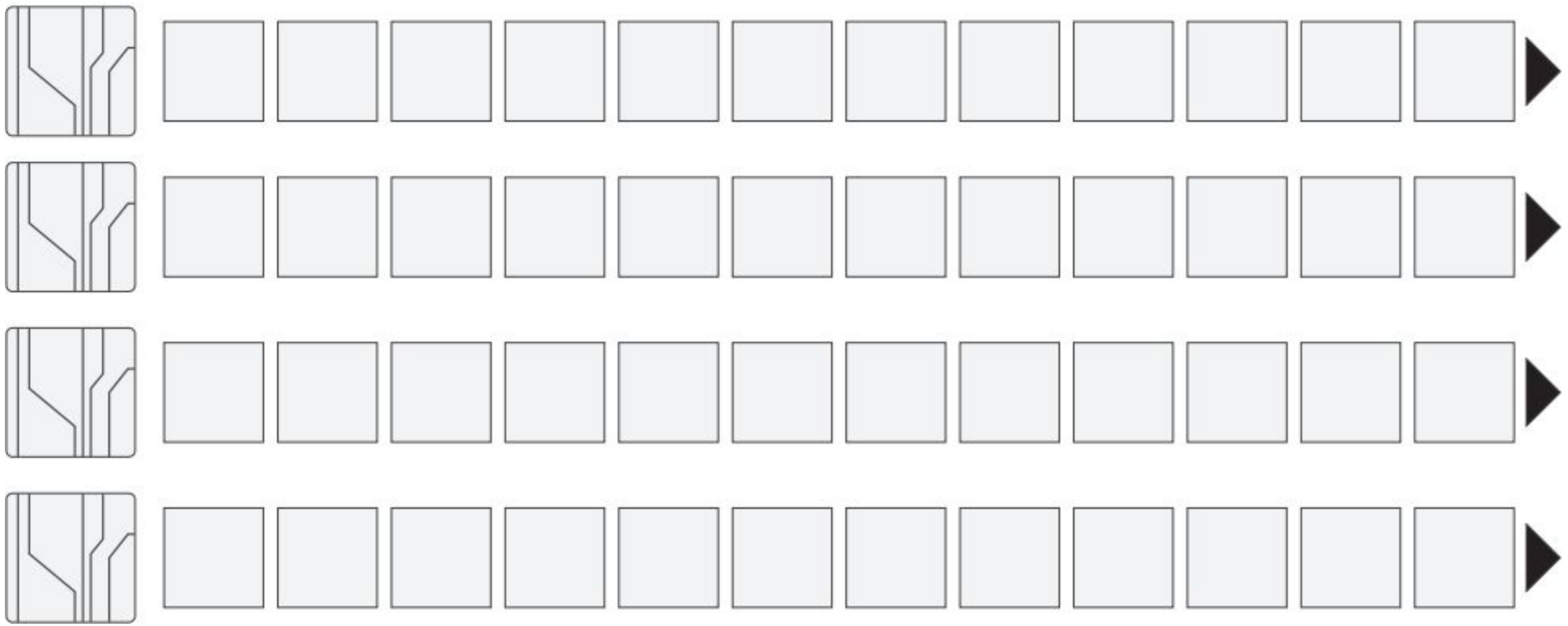


Goal = Save power

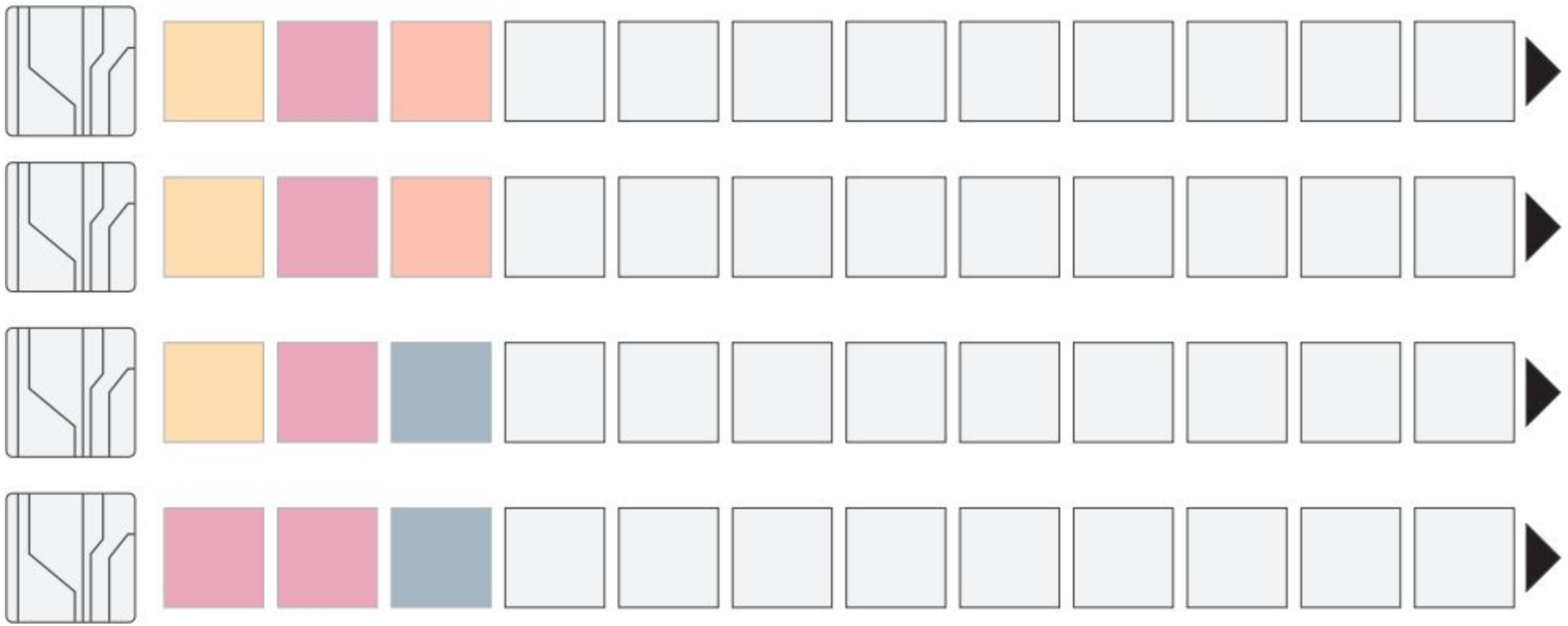
"The minimum **speed** is limited by the **sequential** job model"

- J. Anderson, S. Baruah

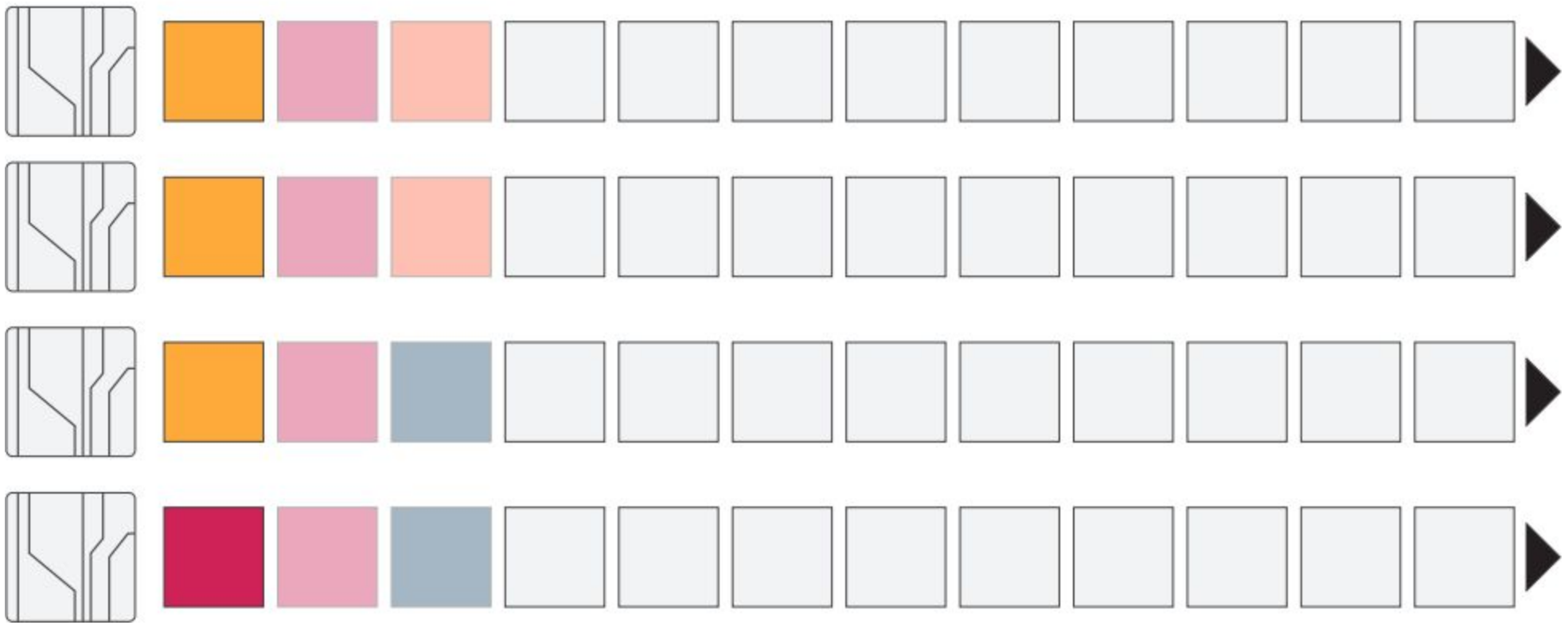
Parallel schedule



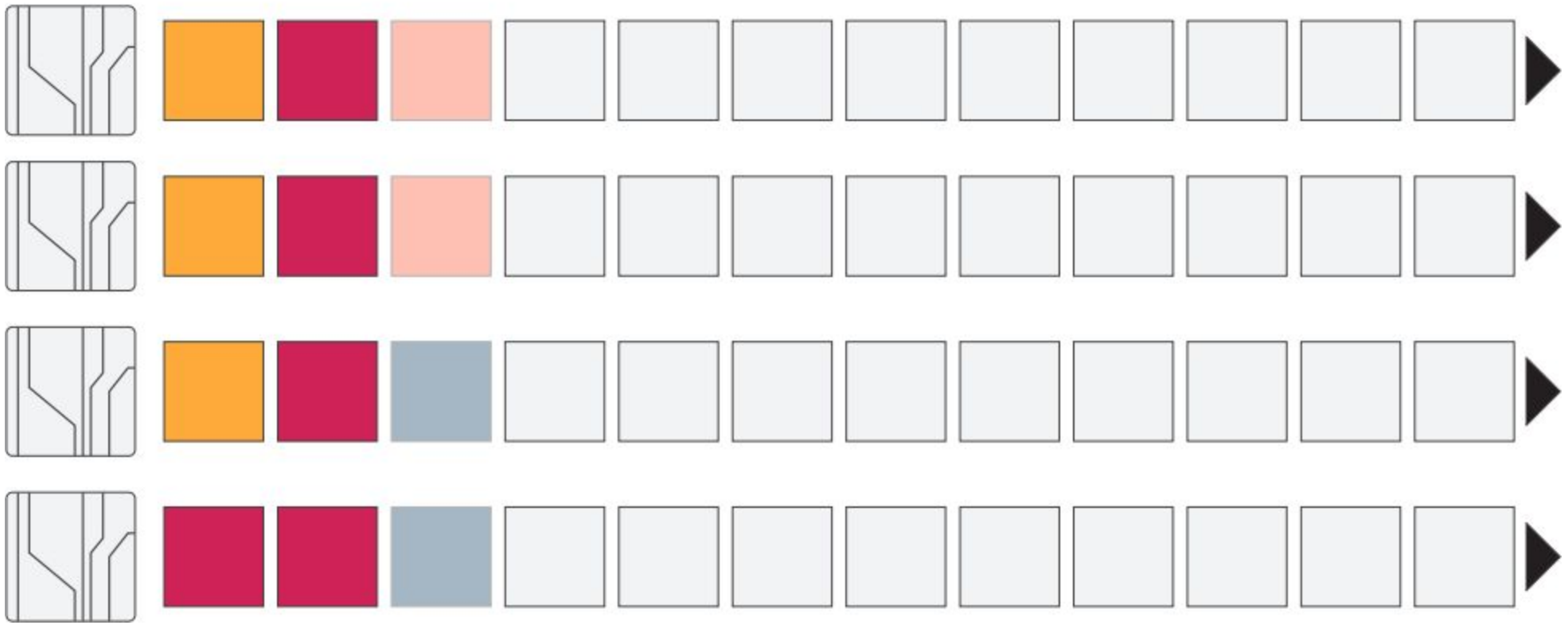
Parallel schedule



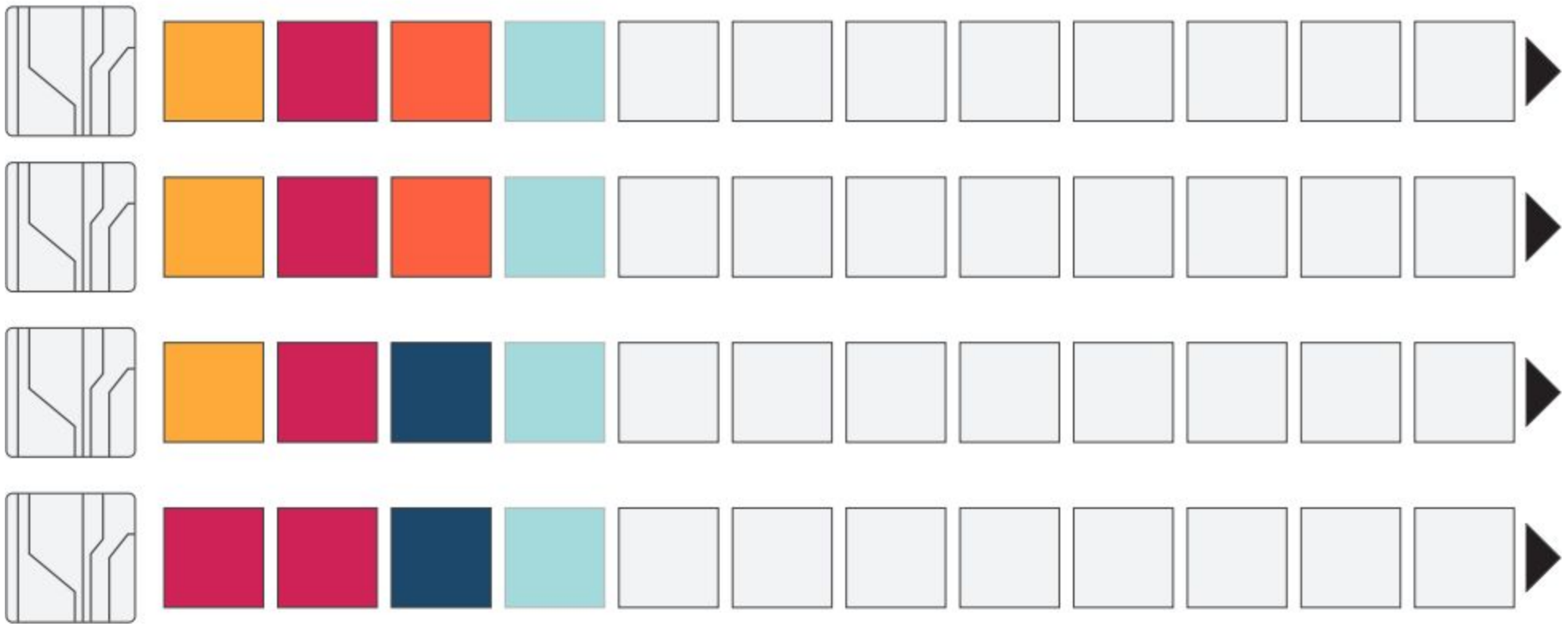
Parallel schedule



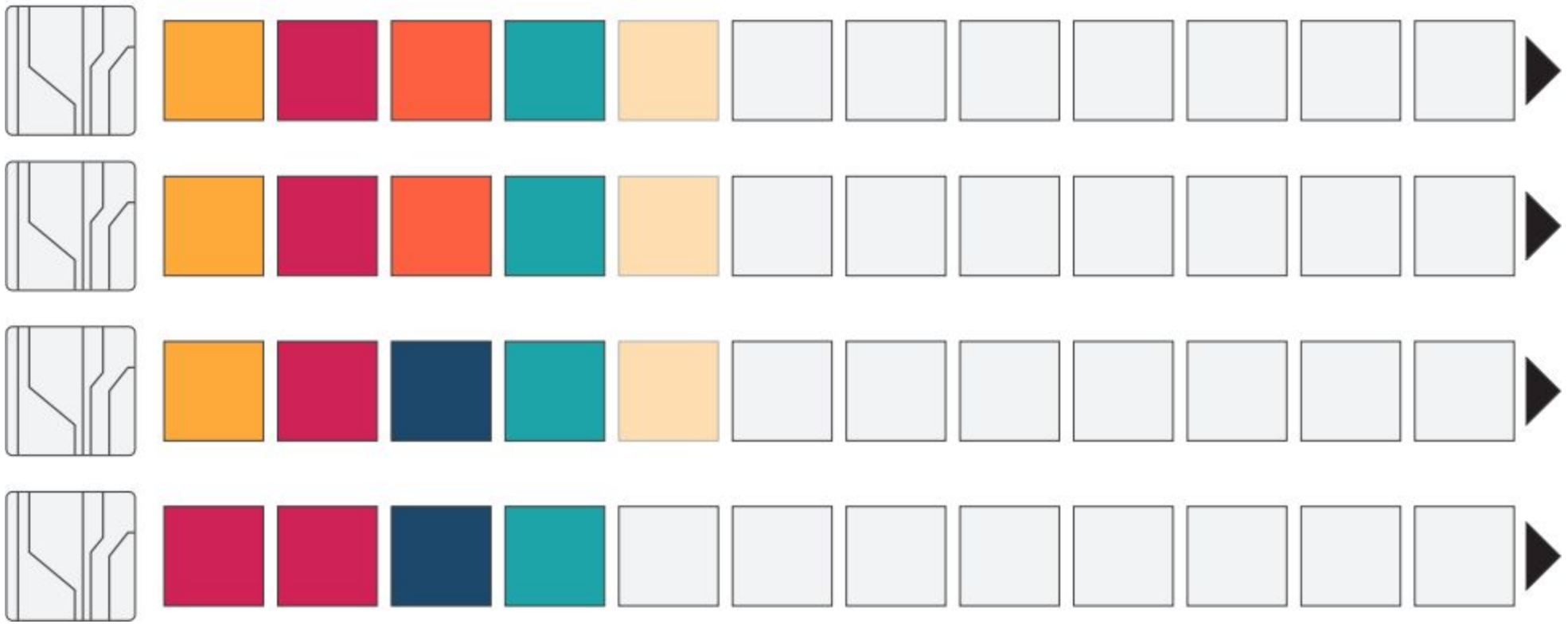
Parallel schedule



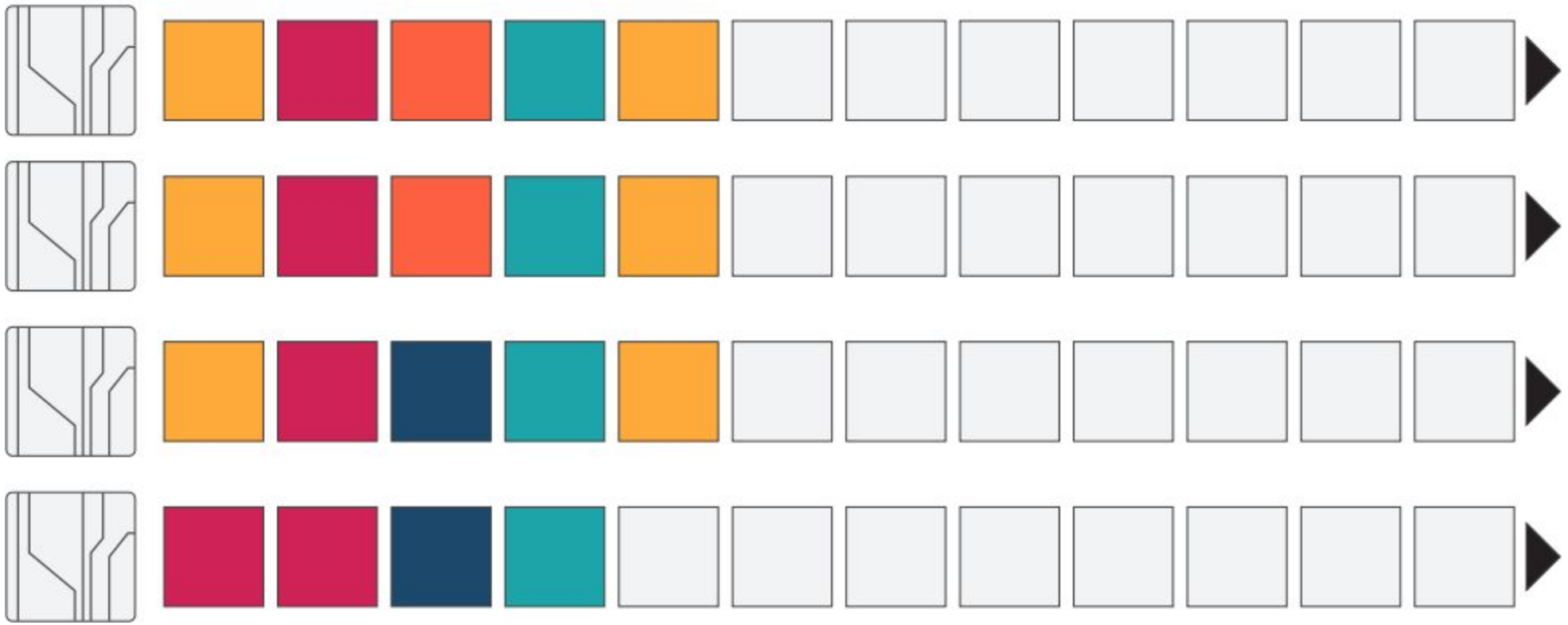
Parallel schedule



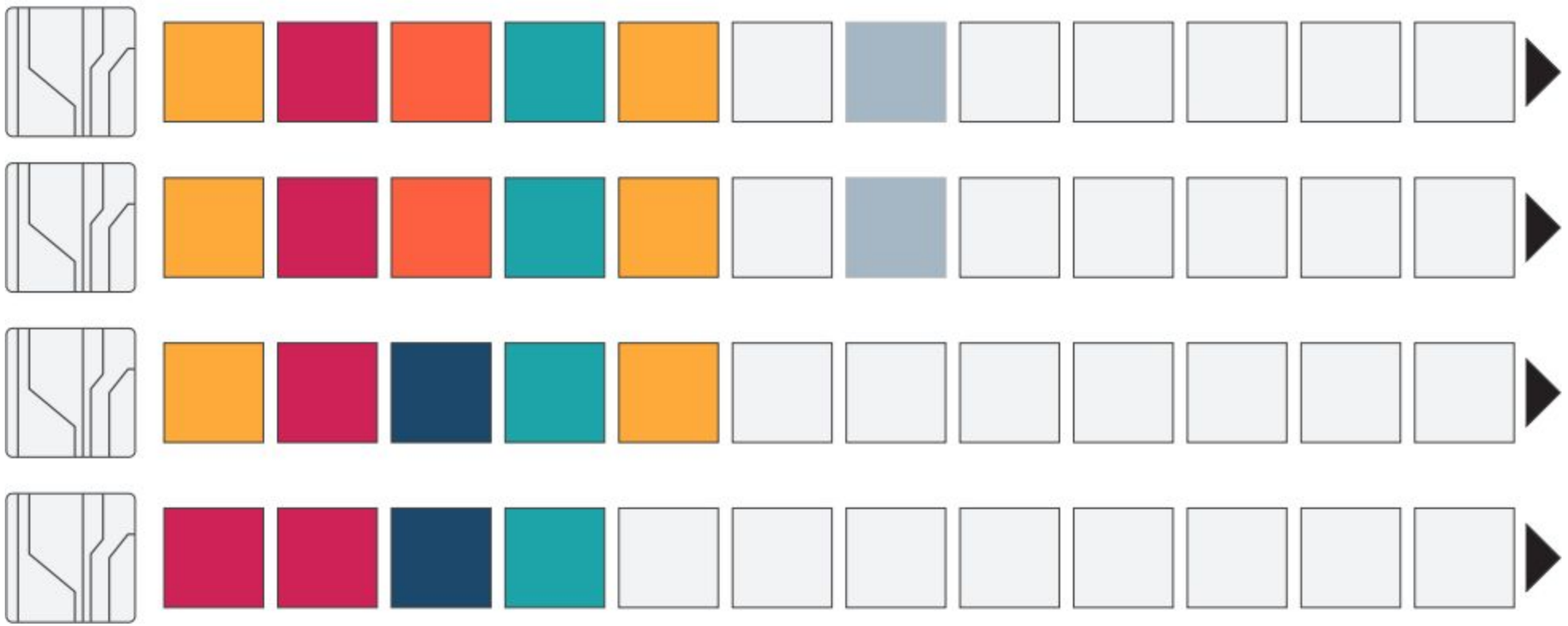
Parallel schedule



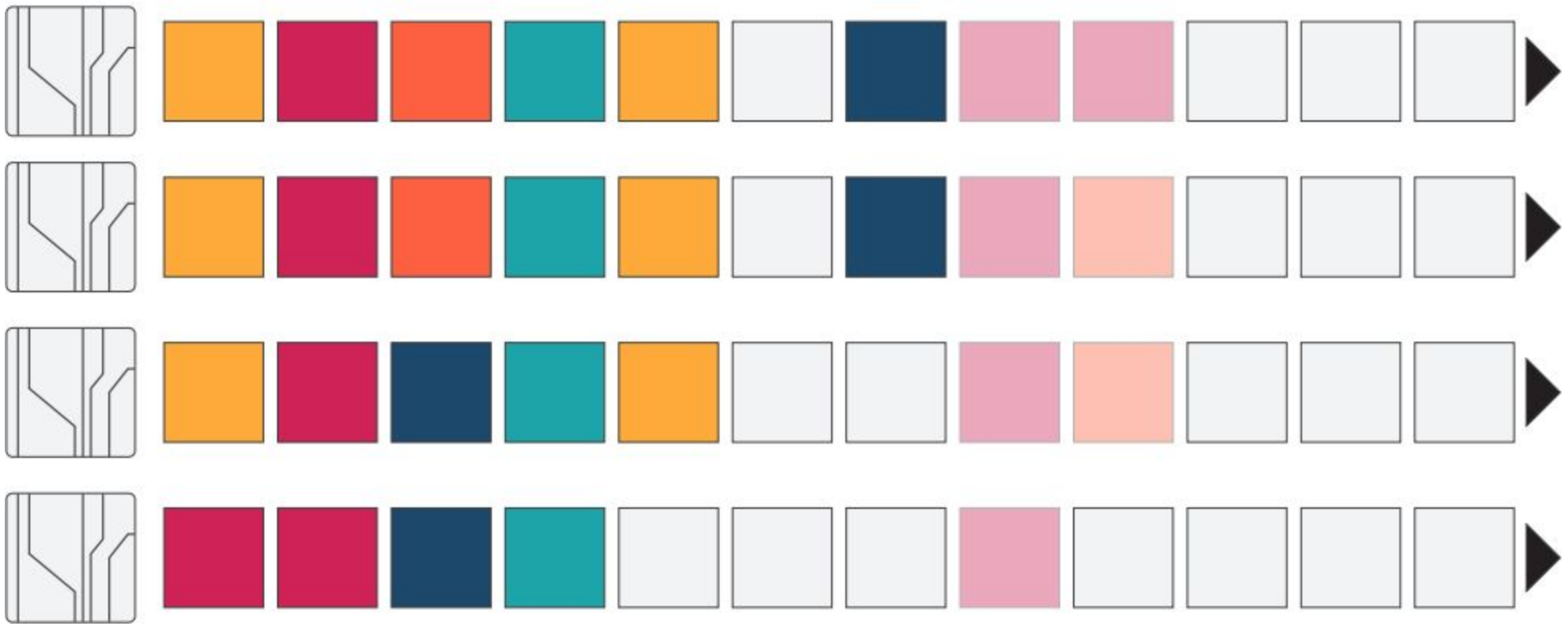
Parallel schedule



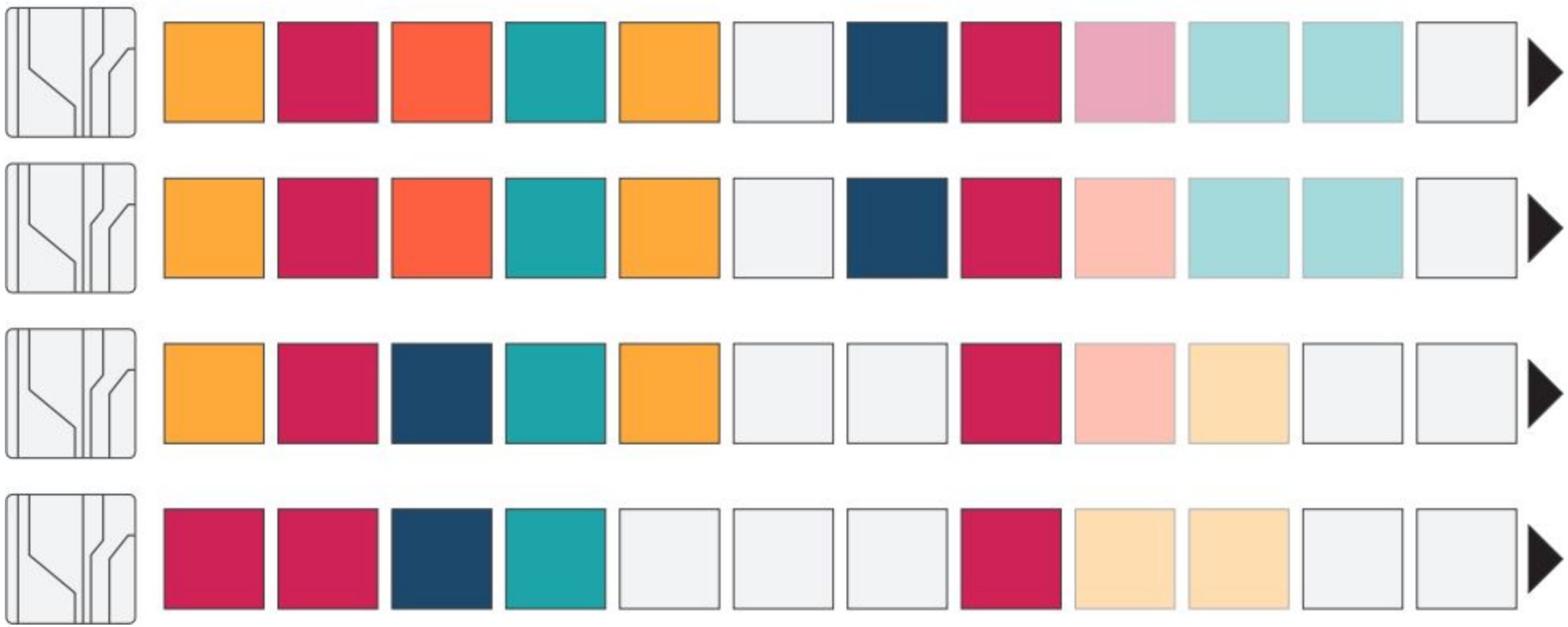
Parallel schedule



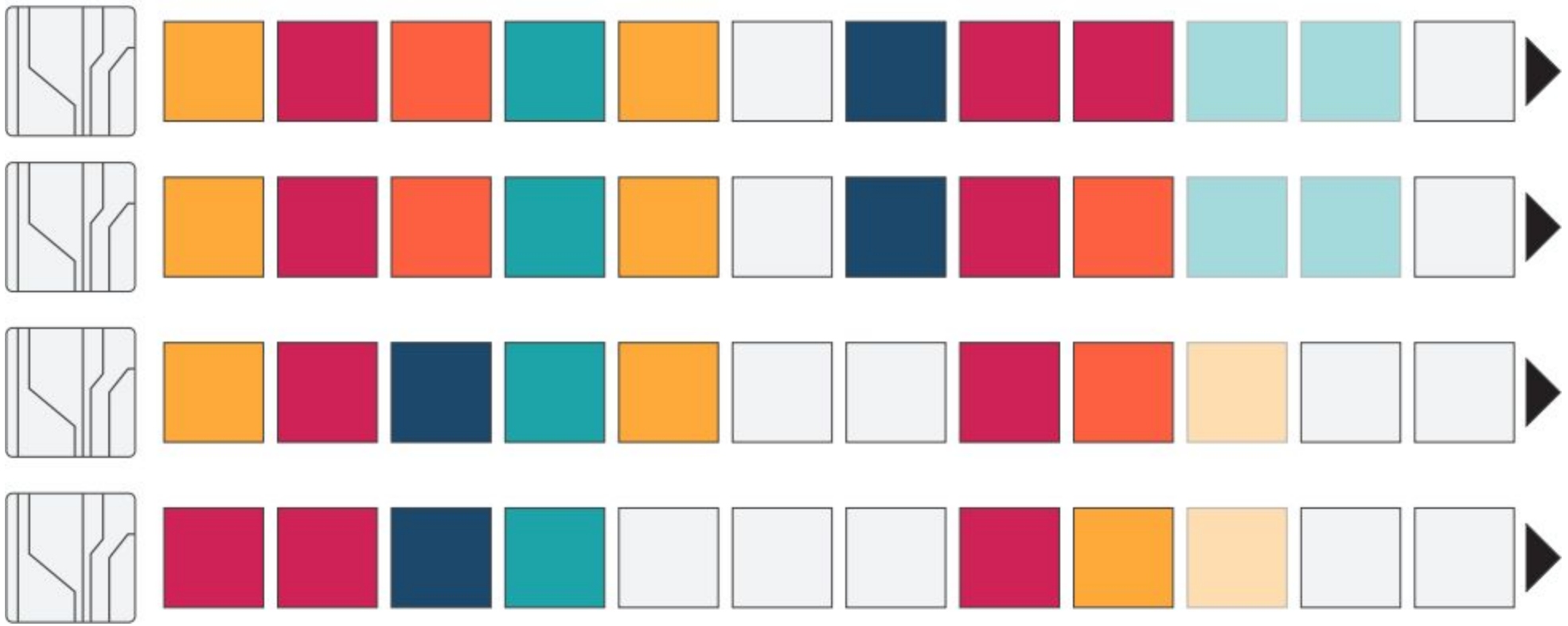
Parallel schedule



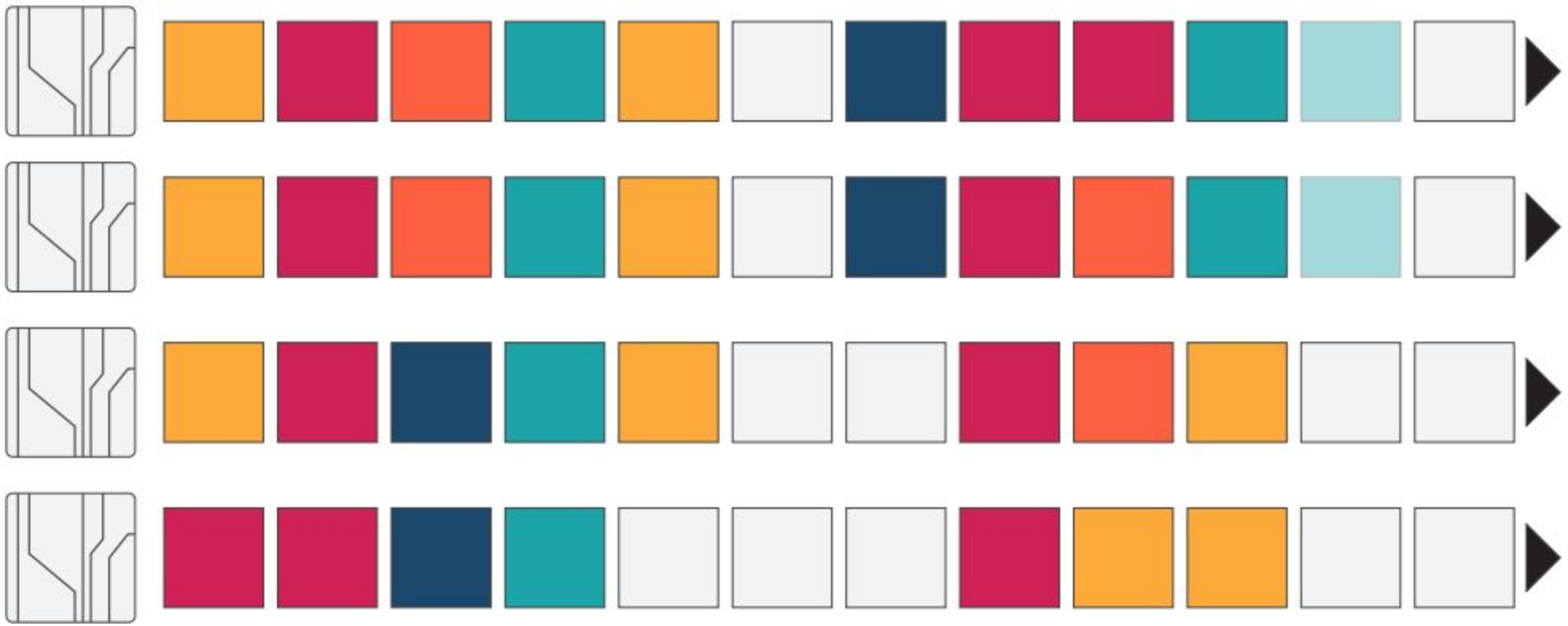
Parallel schedule



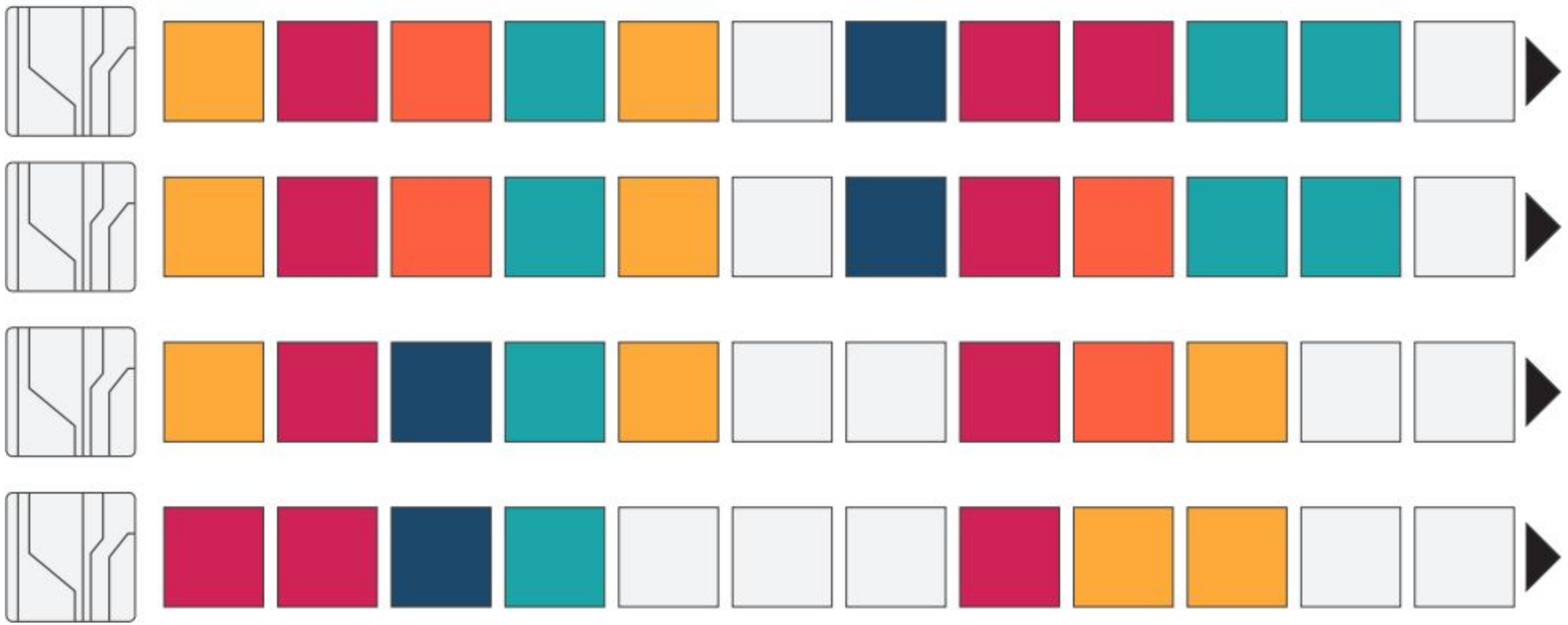
Parallel schedule



Parallel schedule



Parallel schedule



Can we save **more**
with **parallelism**?

Save more with parallelism? Yes!

Combine:

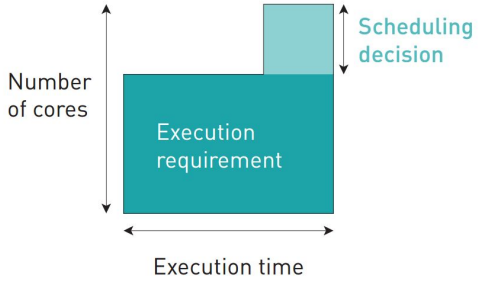
- parallel task model
- Power-aware multi-core scheduling

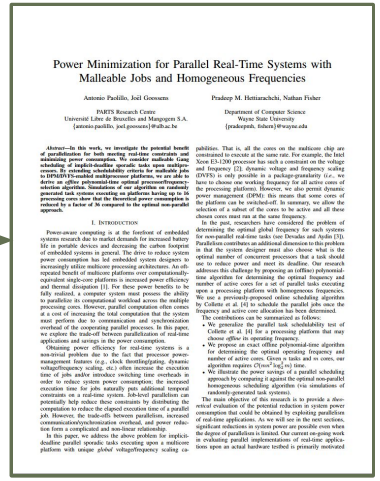
Save more with parallelism? Yes!

Combine:

- parallel task model
- Power-aware multi-core scheduling

→ validated in **theory** → **malleable** jobs (RTCSA'14)




Power Minimization for Parallel Real-Time Systems with Malleable Jobs and Homogeneous Frequencies

Antonio Paolillo, Joël Goossens
PARTS Research Centre
Université Libre de Bruxelles and Mapgyver S.A.
{antonio.paolillo, joel.goossens}@ulb.ac.be

Pradeep M. Hettiarachchi, Nathan Fisher
Department of Computer Science
Wayne State University
{pradeepmh, fishern}@wayne.edu

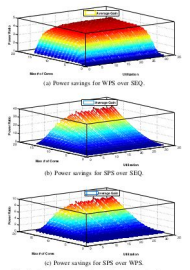


Fig. 4: Average power savings of parallelized systems.

This work

Goal:

Reduce **energy consumption** as much possible
while meeting **real-time requirements**.

This work

Goal:

Reduce **energy consumption** as much possible
while meeting **real-time requirements**.

Validate it ***in practice***

Validate in practice

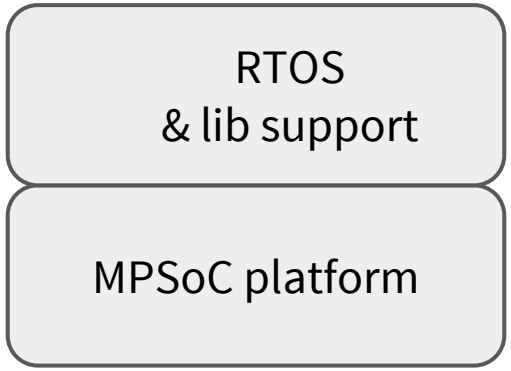
- Parallel programming model
- RTOS techniques → power-aware multi-core hard real-time scheduling
- Evaluated with full experimental stack (RTOS + hardware in the loop)
- Simple, but real application use cases

1. Run-time framework
2. Task model and analysis
3. Experiments

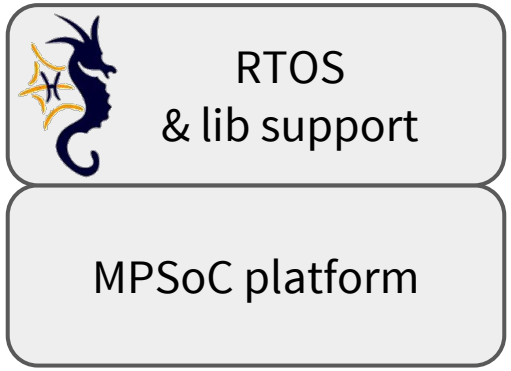
1. Run-time framework
2. Task model and analysis
3. Experiments

Run-time framework

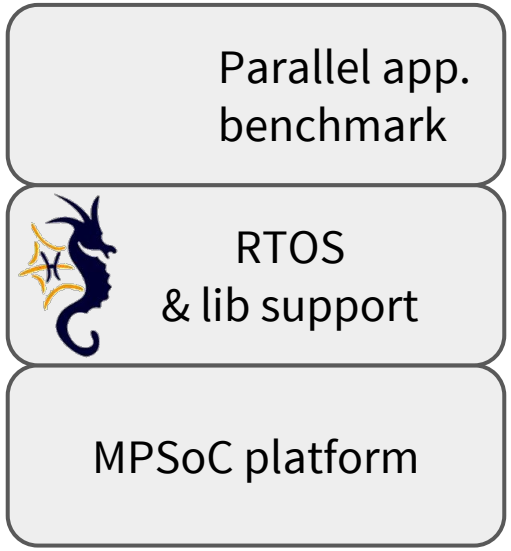
Run-time framework



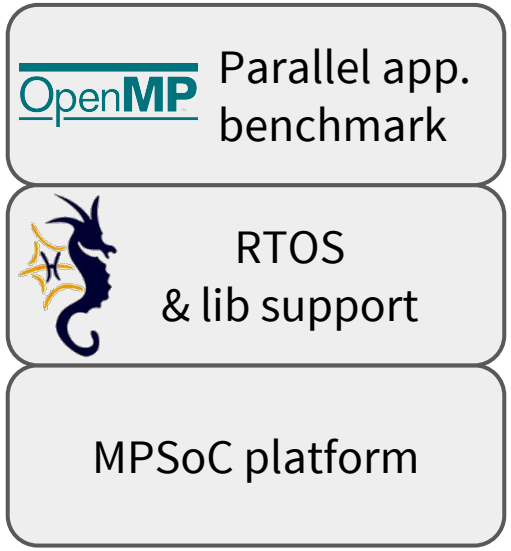
Run-time framework



Run-time framework



Run-time framework

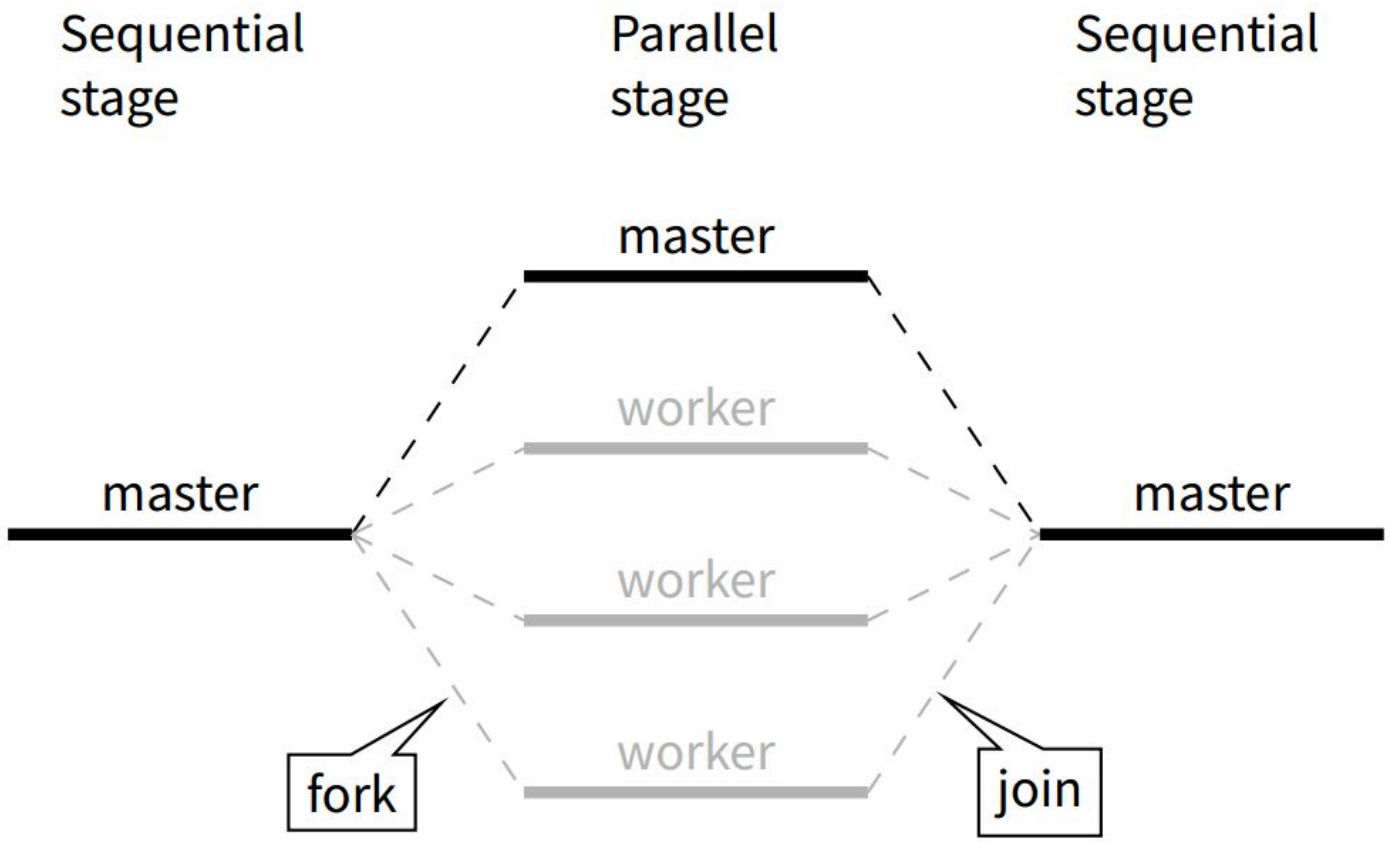


Parallel programming model

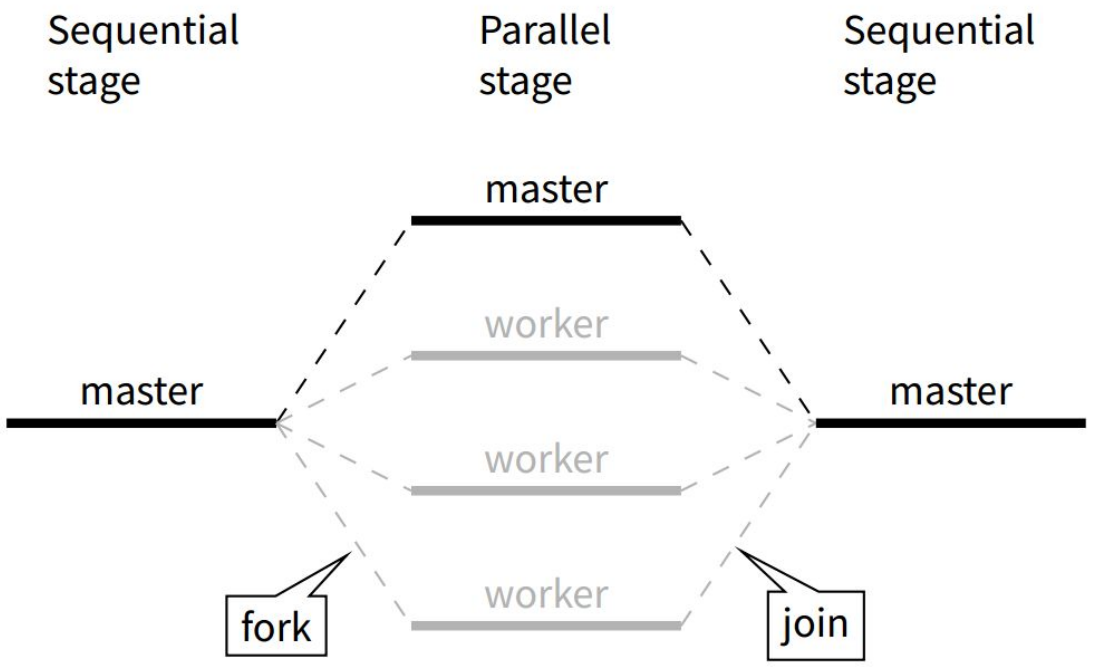
(Simple) Parallel Program

```
1  int main()
2  {
3    const int num_steps = 1000;
4    omp_set_num_threads(4);
5
6  #pragma omp parallel
7    {
8      int nbt = omp_get_num_threads();
9      int tid = omp_get_thread_num();
10     int i_start = (tid * num_steps) / nbt;
11     int i_end = ((tid + 1) * num_steps) / nbt;
12
13     for (int i = i_start; i < i_end; ++i) {
14       /* workload executed in parallel */
15     }
16   }
17
18   return 0;
19 }
```

(Simple) Parallel Program Flow



(Simple) Parallel Program Flow




```
1  int main()
2  {
3    const int num_steps = 1000;
4    omp_set_num_threads(4);
5
6  #pragma omp parallel
7    {
8      int nbt = omp_get_num_threads();
9      int tid = omp_get_thread_num();
10     int i_start = (tid * num_steps) / nbt;
11     int i_end = ((tid + 1) * num_steps) / nbt;
12
13     for (int i = i_start; i < i_end; ++i) {
14       /* workload executed in parallel */
15     }
16   }
17
18   return 0;
19 }
```

An embedded RTOS

An embedded RTOS

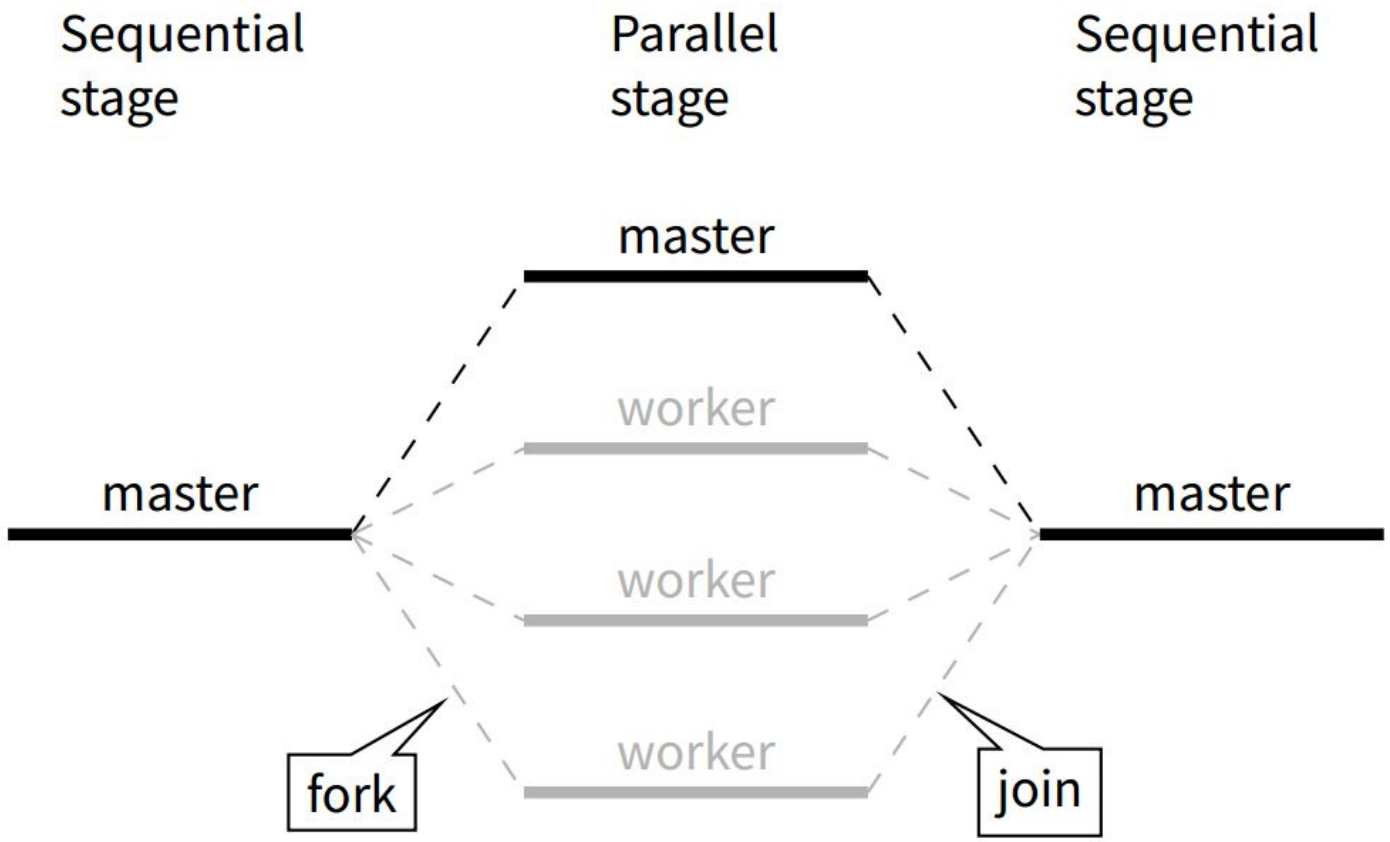
- Micro-kernel based
- Natively supports multi-core
- Provides hard-real time scheduling
- Power management fixed at task level

We developed HOMPRTL

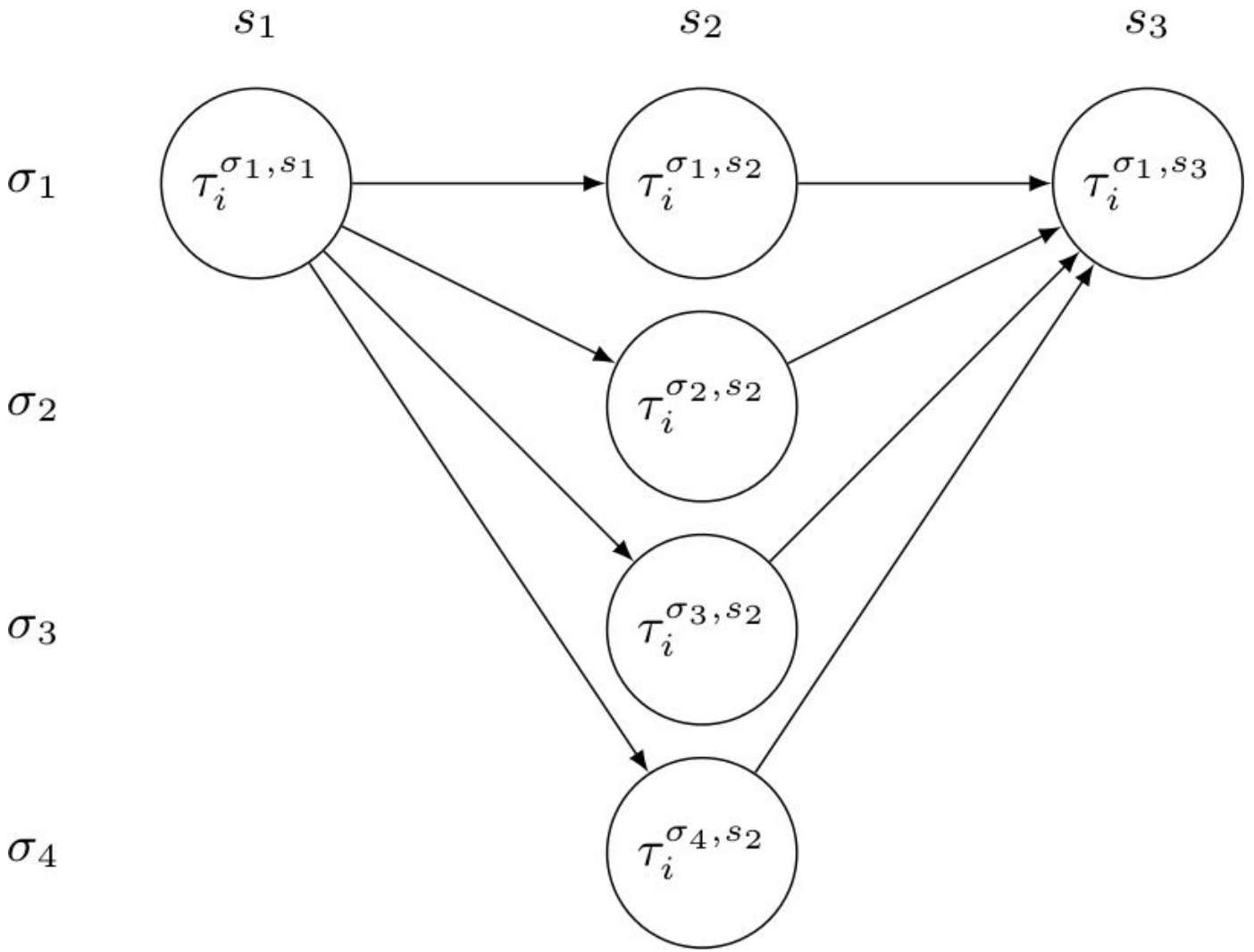
→ OpenMP run-time support for HIPPEROS.

1. Run-time framework
2. Task model and analysis
3. Experiments

Parallel task model

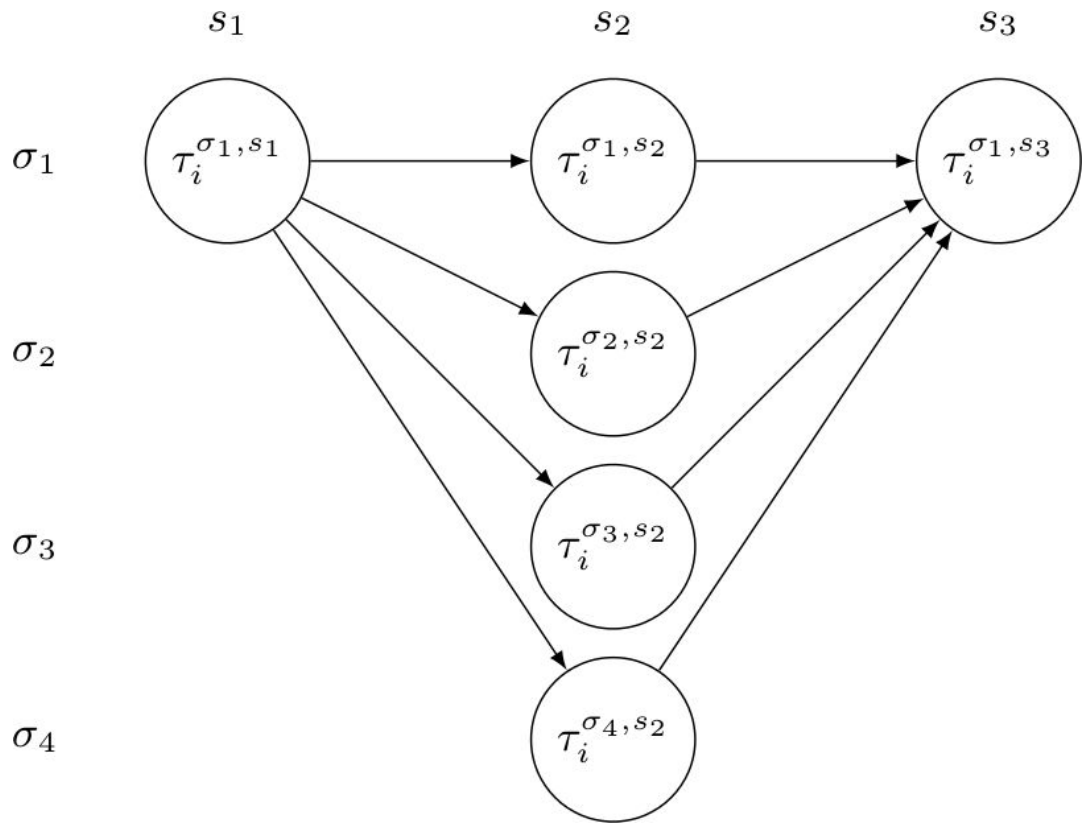


Parallel task model



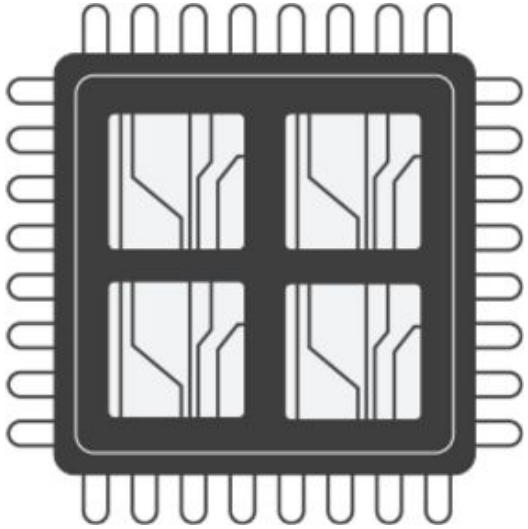
Parallel task model

- Sporadic fork-join
- Arbitrary deadlines
- Rate monotonic based analysis
- Threads are partitioned
- 3 stages, very simple

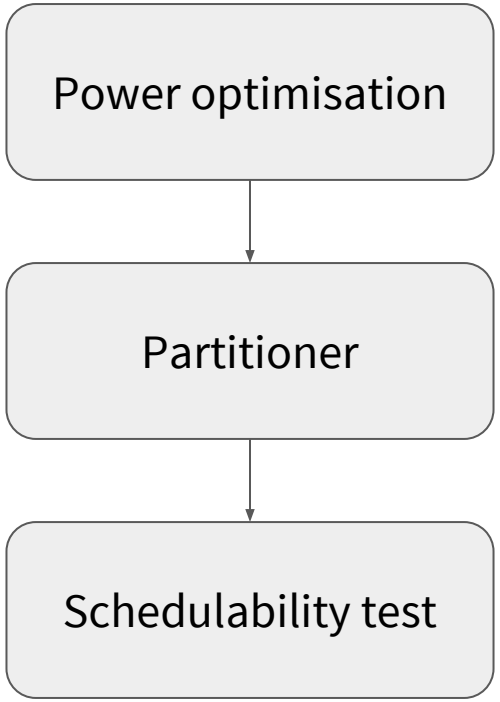


Platform model

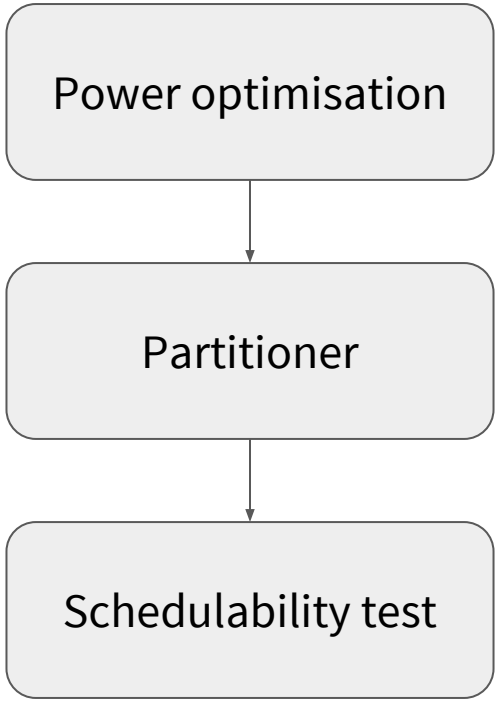
- Symmetric Multi-Core
- Global DVFS, finite set of operating points $\langle V_{dd}, f \rangle$
- Power increases monotonically with frequency



Optimisation & partitioning process



Optimisation & partitioning process



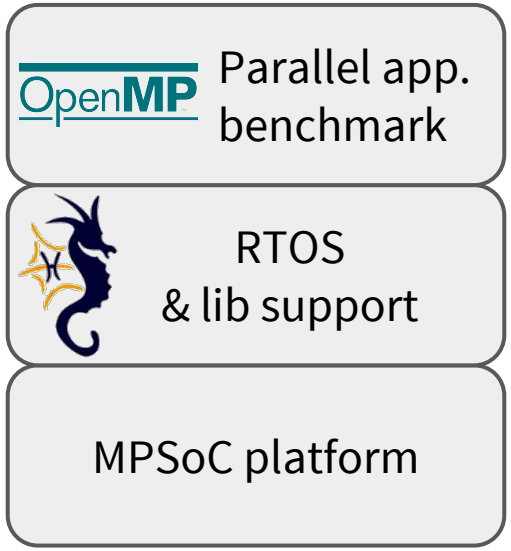
Axer et al
proved wrong (shepherding)

1. Run-time framework
2. Task model and analysis
3. Experiments

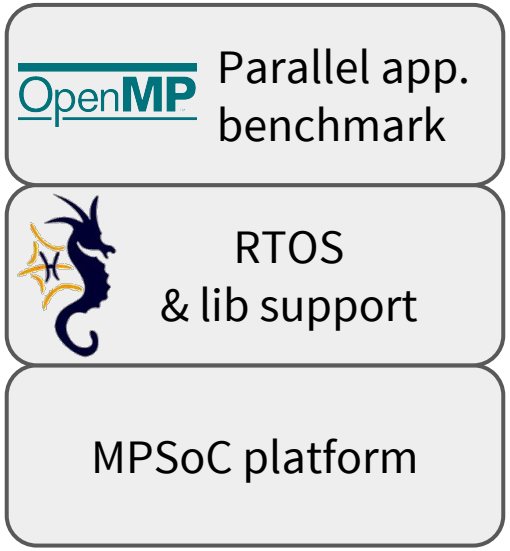
1. Run-time framework
2. Task model and analysis
3. Experiments
    a. Testbed description
    b. Single use cases
    c. Task systems

1. Run-time framework
2. Task model and analysis
3. Experiments
    a. Testbed description
    b. Single use cases
    c. Task systems

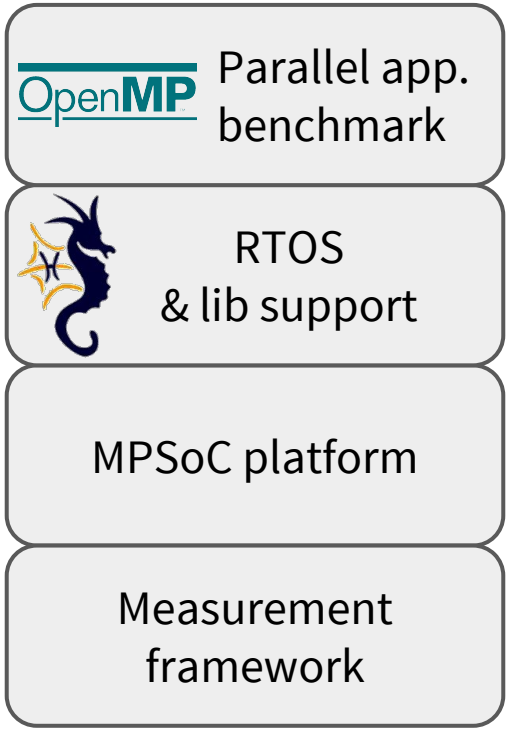
Experimental stack



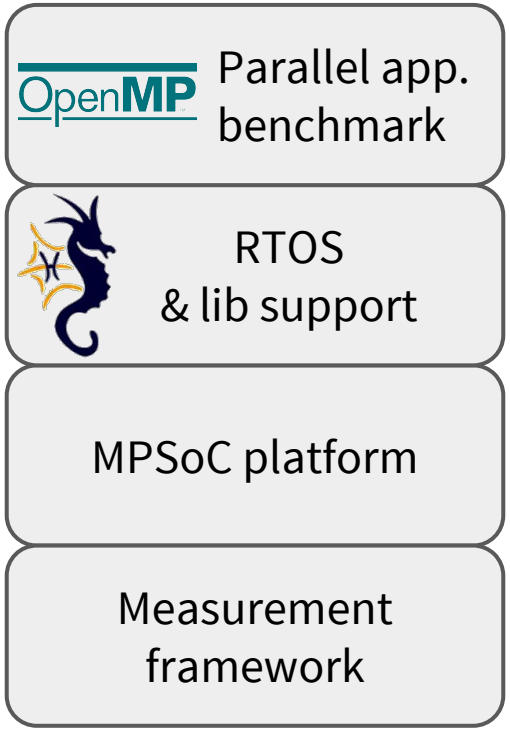
Experimental stack



Experimental stack



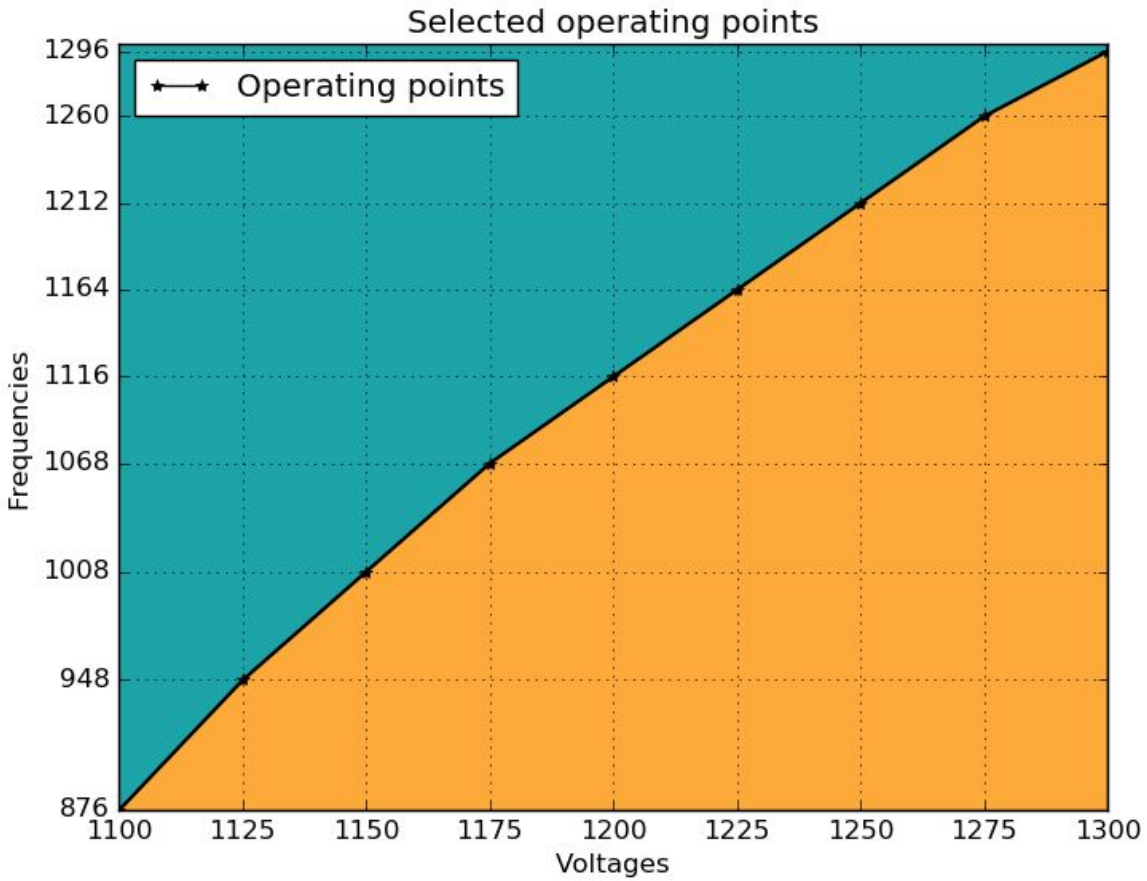
Experimental stack



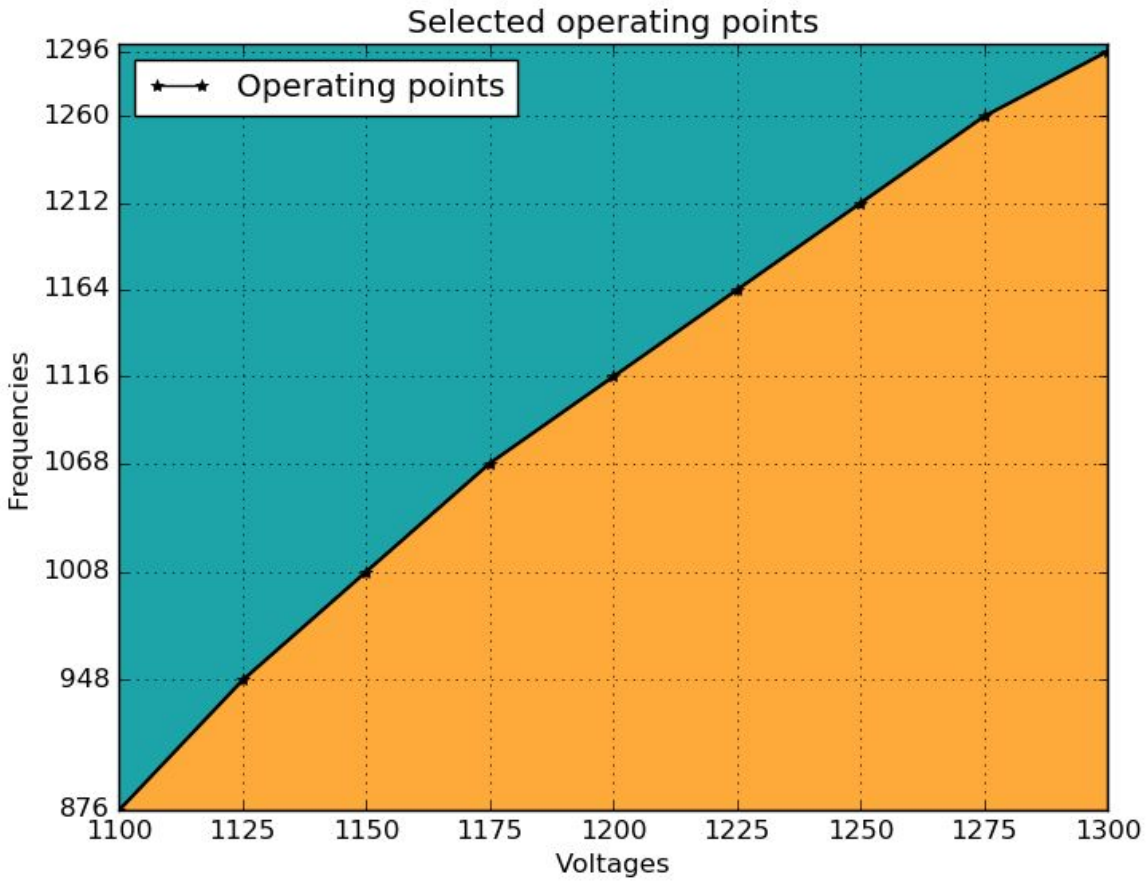
Target platform - i.MX6q SabreLite

- Embedded board ARM Cortex A9 MP
- Supported by HIPPEROS
- 4 cores, but global DVFS
- Operating points

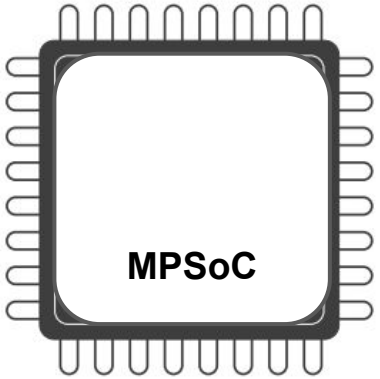
Operating points



Operating points

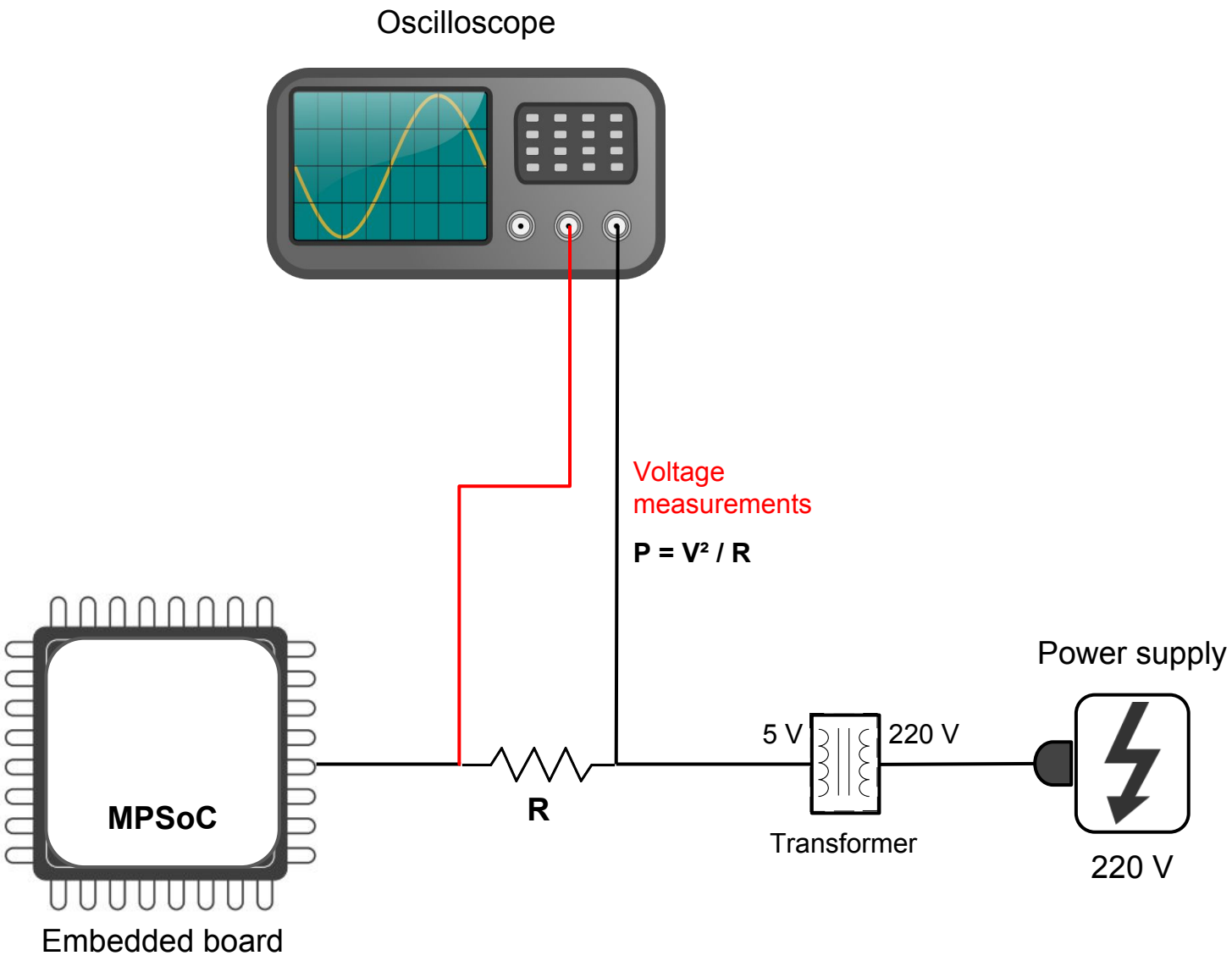


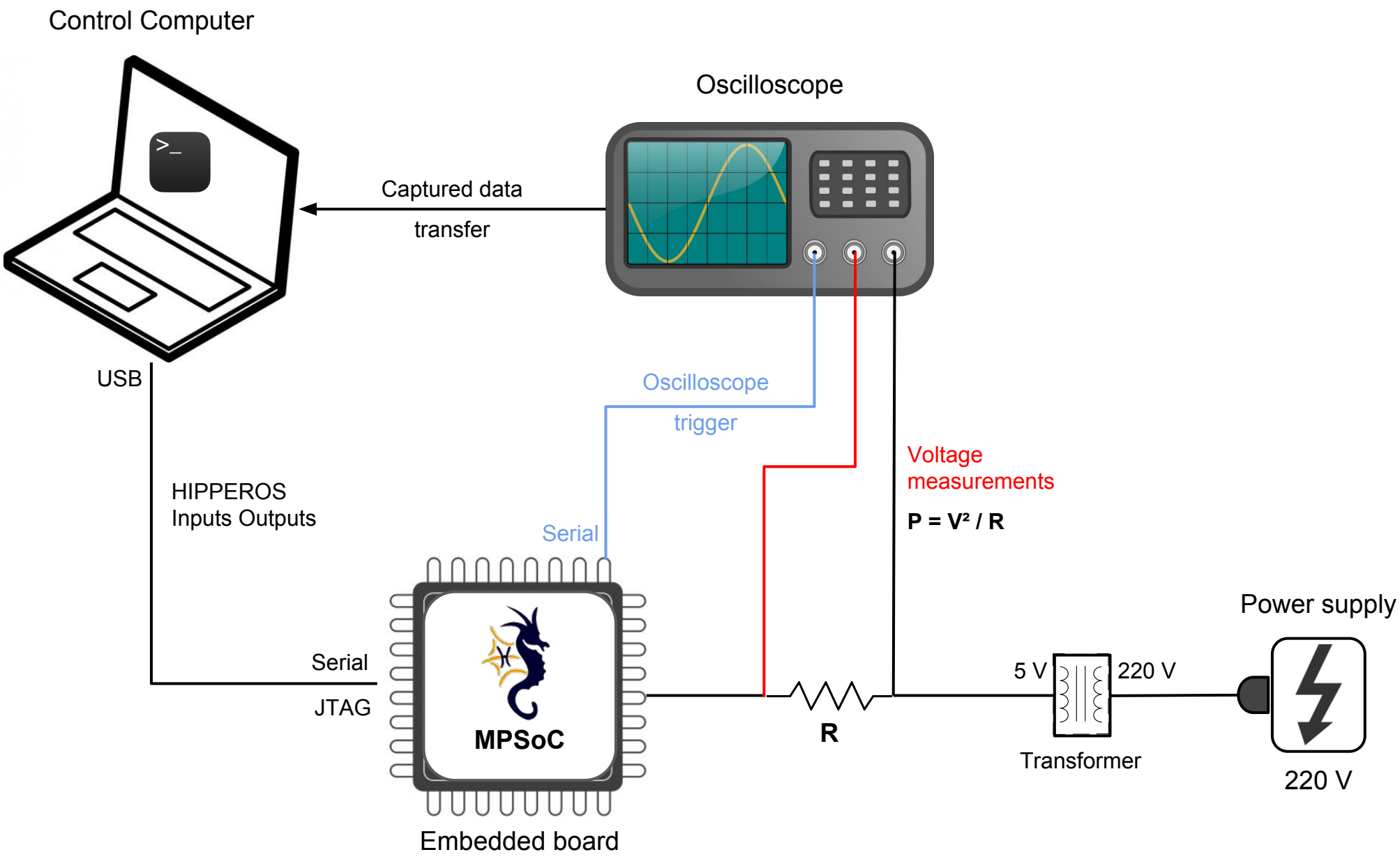
Voltage (mV)	Frequency (MHz)
1100	876
1125	948
1150	1008
1175	1068
1200	1116
1225	1164
1250	1212
1275	1260
1300	1296



Embedded board


MPSoC
Embedded board
5 V
220 V
Transformer
Power supply
220 V


Oscilloscope
Voltage
measurements
P = V² / R
Power supply
5 V
220 V
MPSoC
R
Transformer
220 V
Embedded board


Control Computer
Oscilloscope
Captured data
transfer
USB
Oscilloscope
trigger
Voltage
measurements
P = V² / R
HIPPEROS
Inputs Outputs
Serial
Serial
JTAG
MPSoC
R
5 V
220 V
Transformer
Power supply
220 V
Embedded board

Use cases

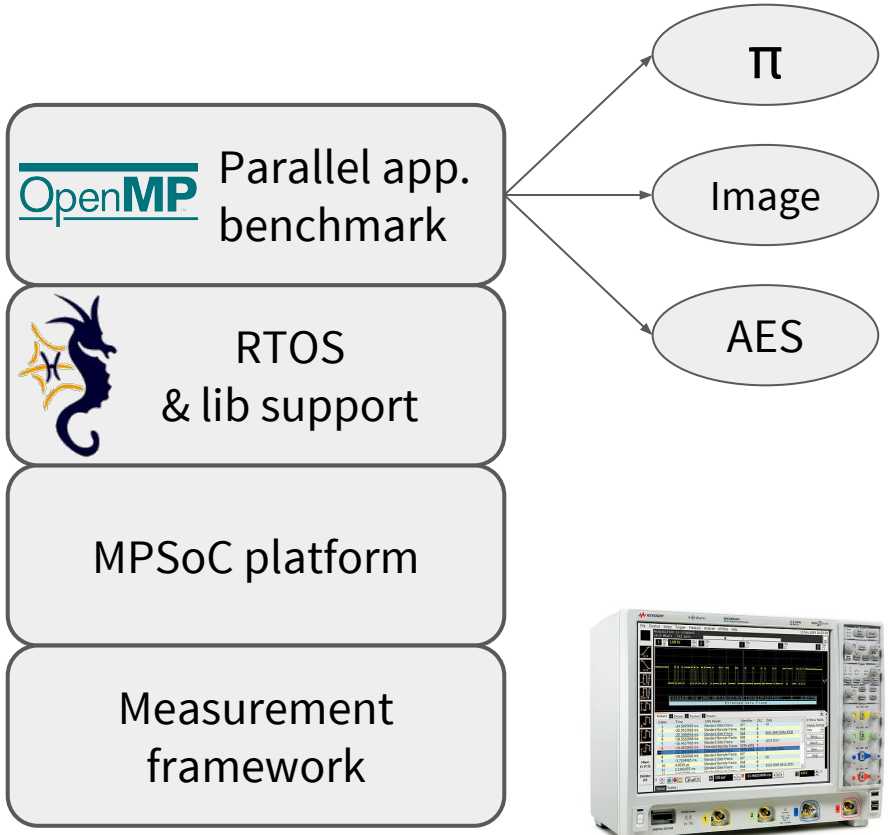
Parallel app. benchmark

RTOS & lib support

MPSoC platform

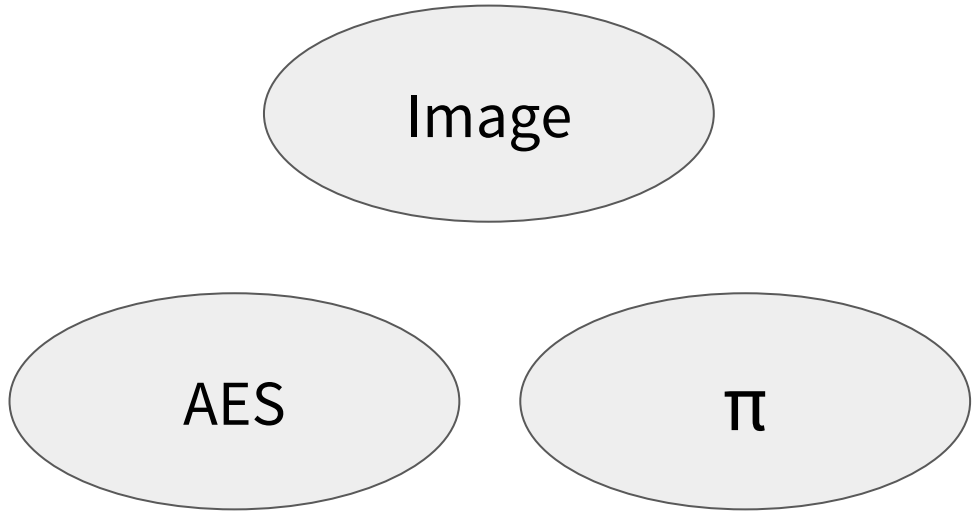
Measurement framework

Use cases


OpenMP
Parallel app. benchmark
RTOS & lib support
MPSoC platform
Measurement framework
π
Image
AES

Use cases

- Different workloads
- Easy OpenMP implementation
- Good scaling expected

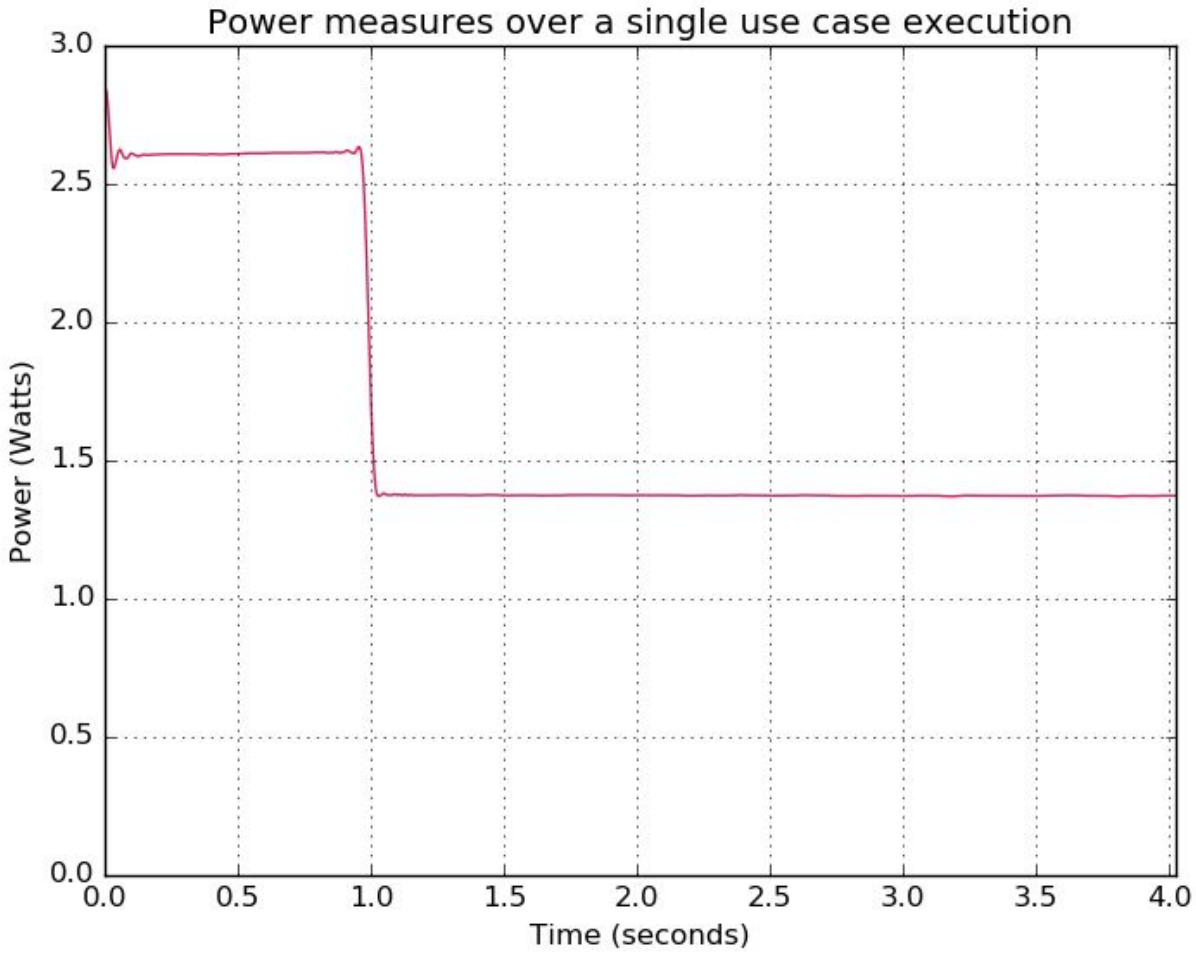



```
int main()
{
  const int num_steps = 1000;
  omp_set_num_threads(4);

#pragma omp parallel
  {
    int nbt = omp_get_num_threads();
    int tid = omp_get_thread_num();
    int i_start = (tid * num_steps) / nbt;
    int i_end = ((tid + 1) * num_steps) / nbt;

    for (int i = i_start; i < i_end; ++i) {
      /* workload executed in parallel */
    }
  }

  return 0;
}
```

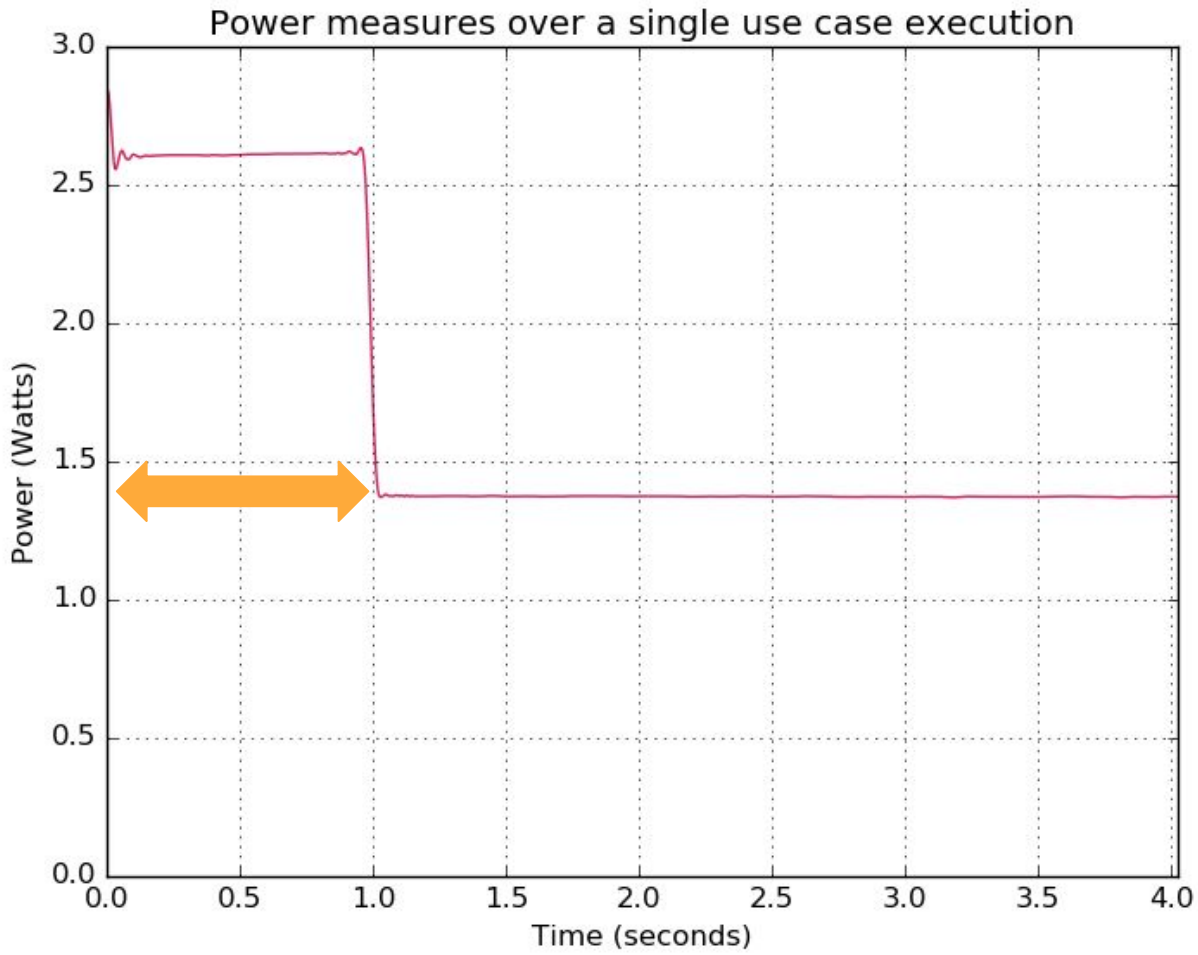
1. Run-time framework
2. Task model and analysis
3. Experiments
    a. Testbed description
    b. Single use cases
    c. Task systems

Use case alone - voltage probe output

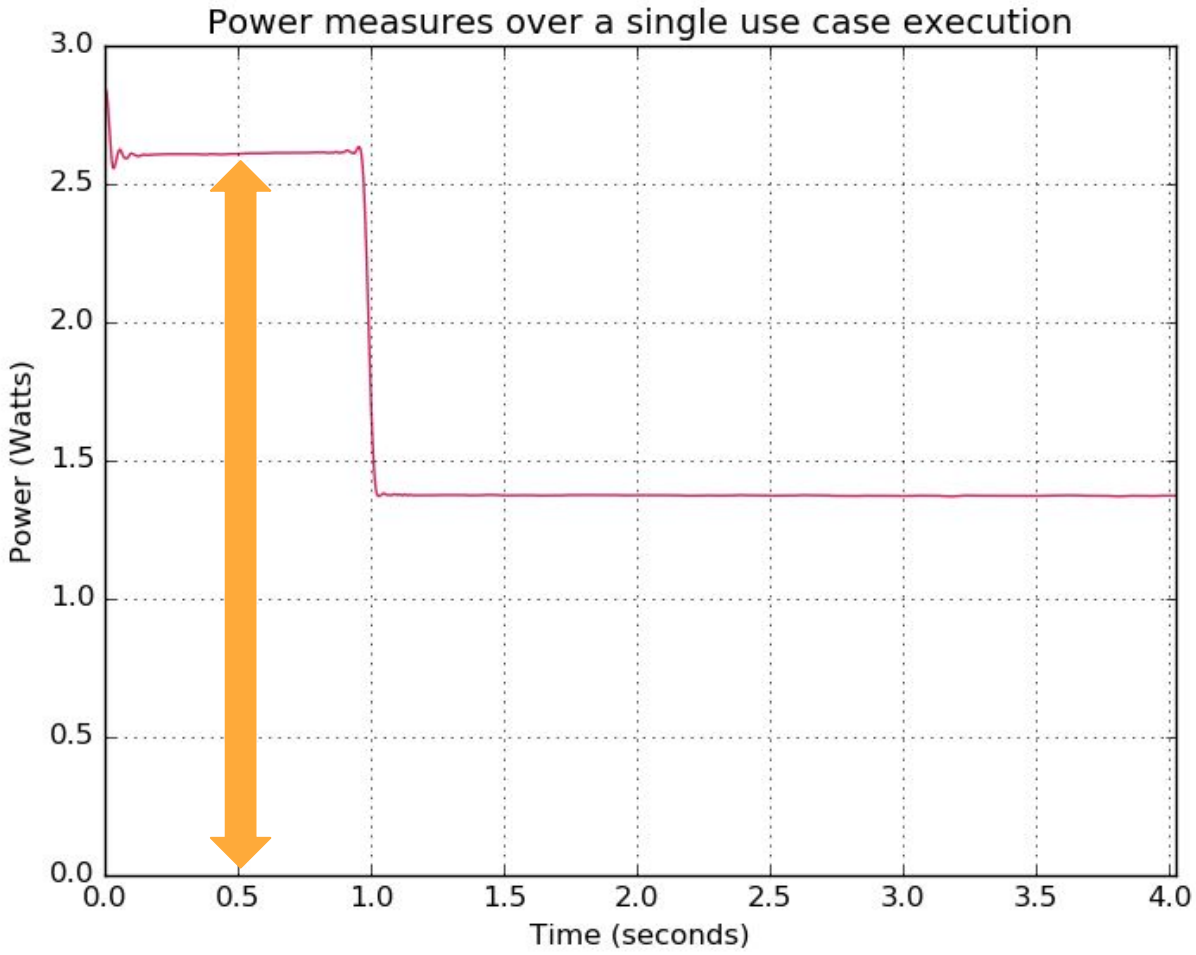
Use case alone - converted to power values



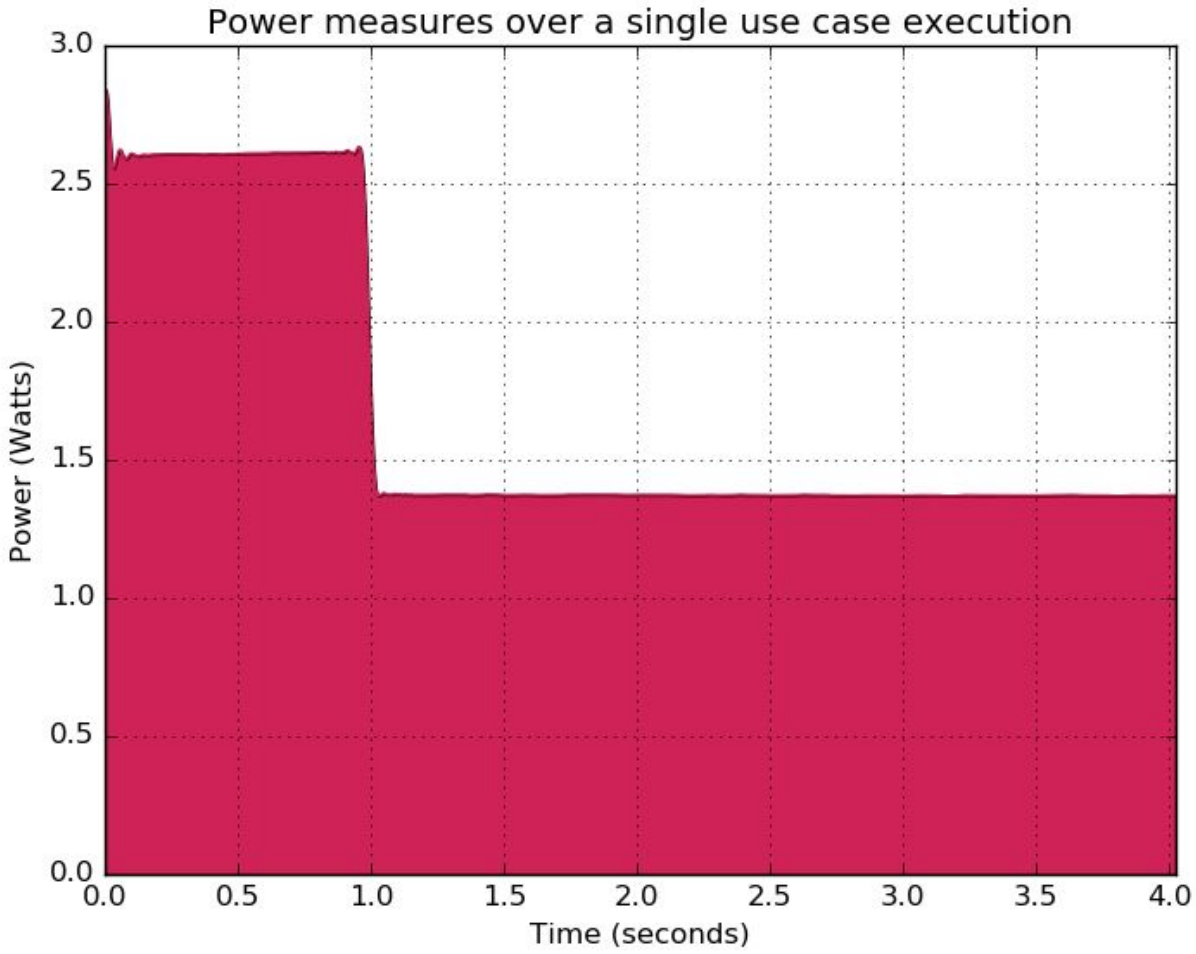
Use case alone - execution time measurement



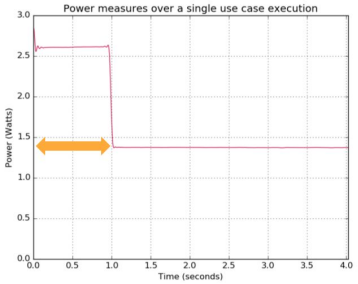
Use case alone - peak power measurement

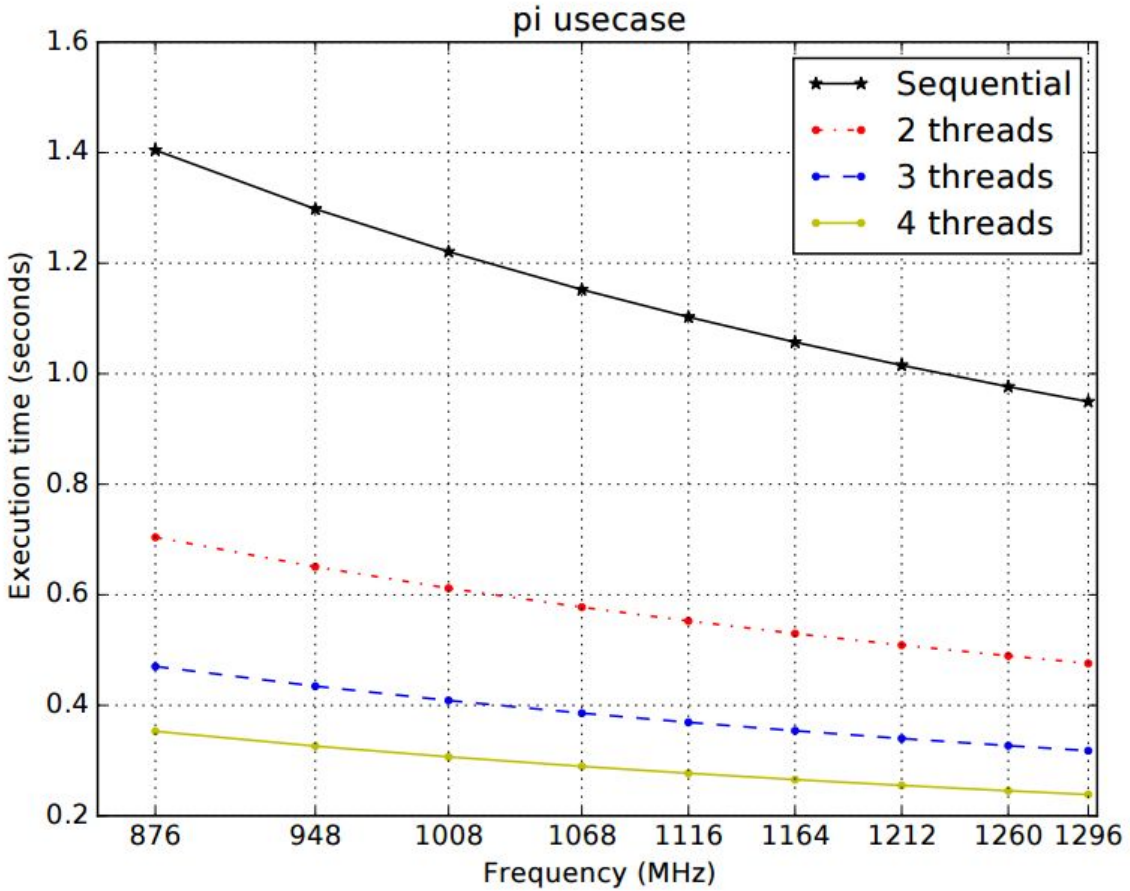


Use case alone - energy measurement

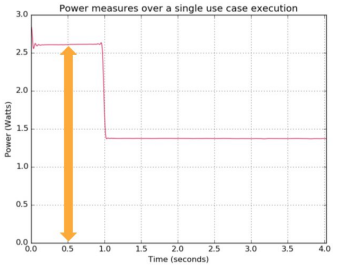


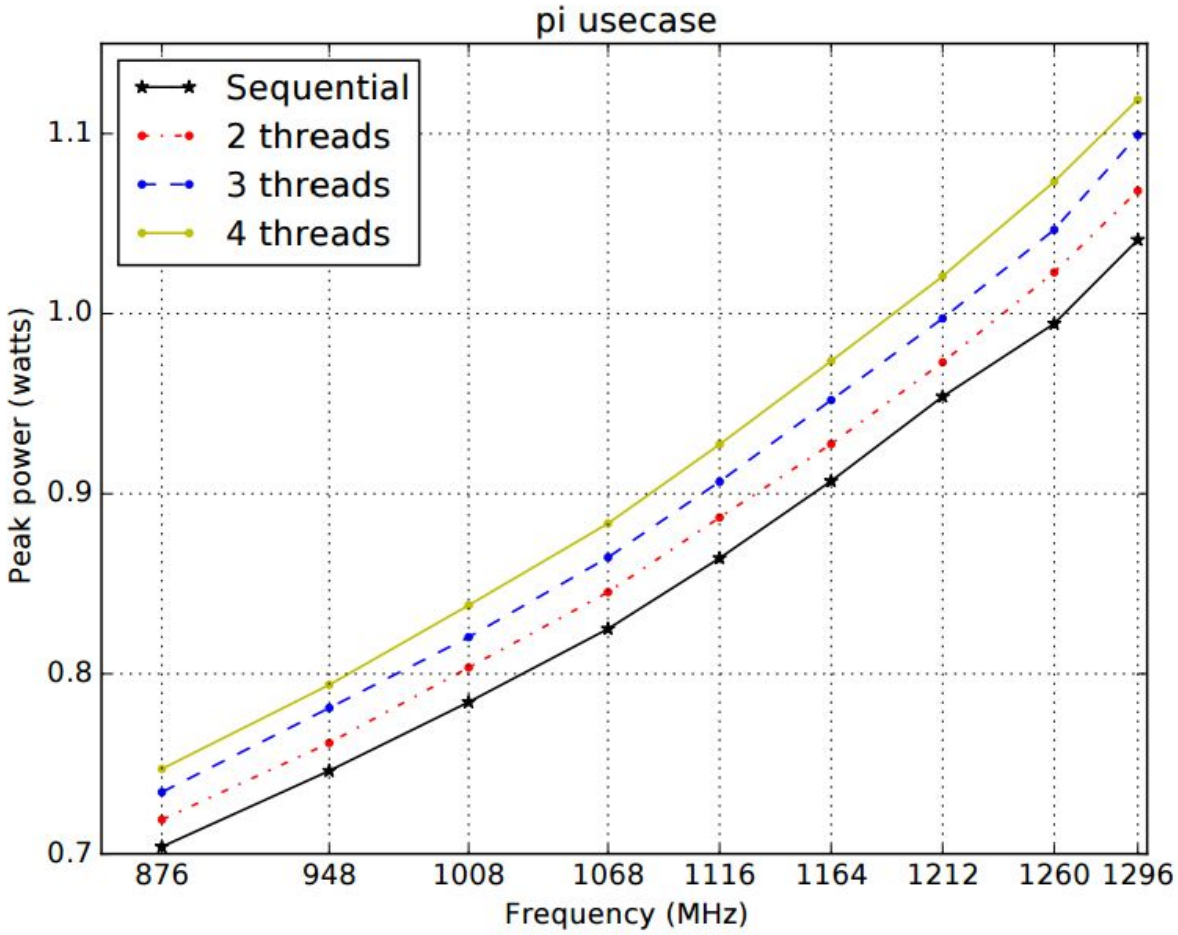
Use case alone - execution time





Use case alone - peak power





Single core power consumption

$$P \propto V_{dd}^{2} f$$

$$V_{dd} \propto f$$

Single core power consumption

$$P \propto V_{dd}^{2} f$$

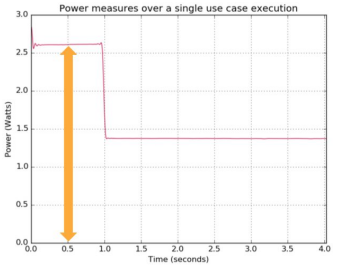
$$V_{dd} \propto f$$

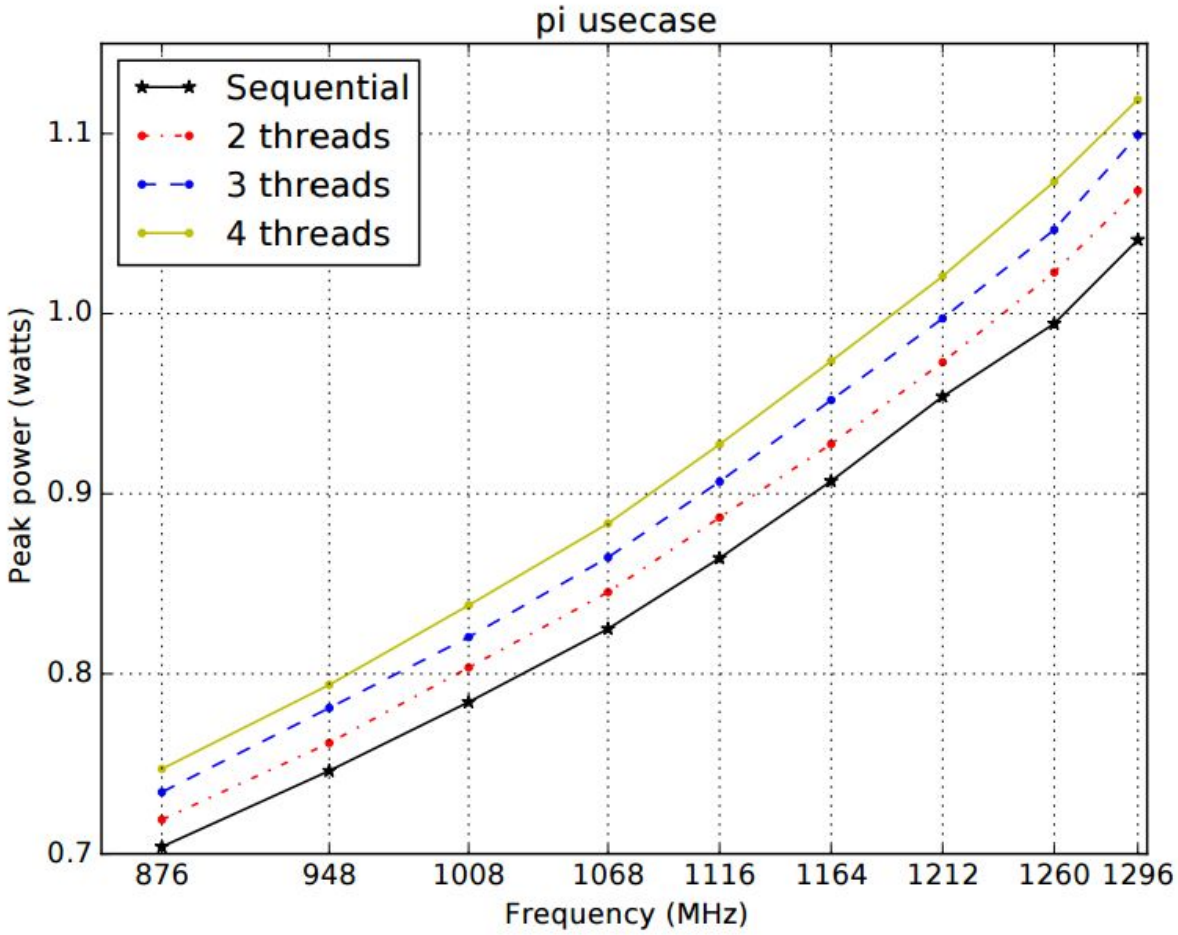
$$\rightarrow P \propto f^{3}$$

Multi-core power consumption

$$P \propto k\ f^3$$

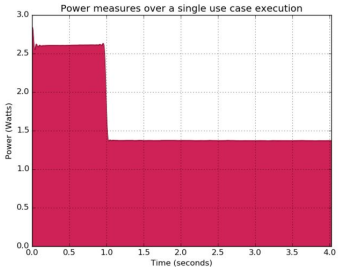
Use case alone - peak power

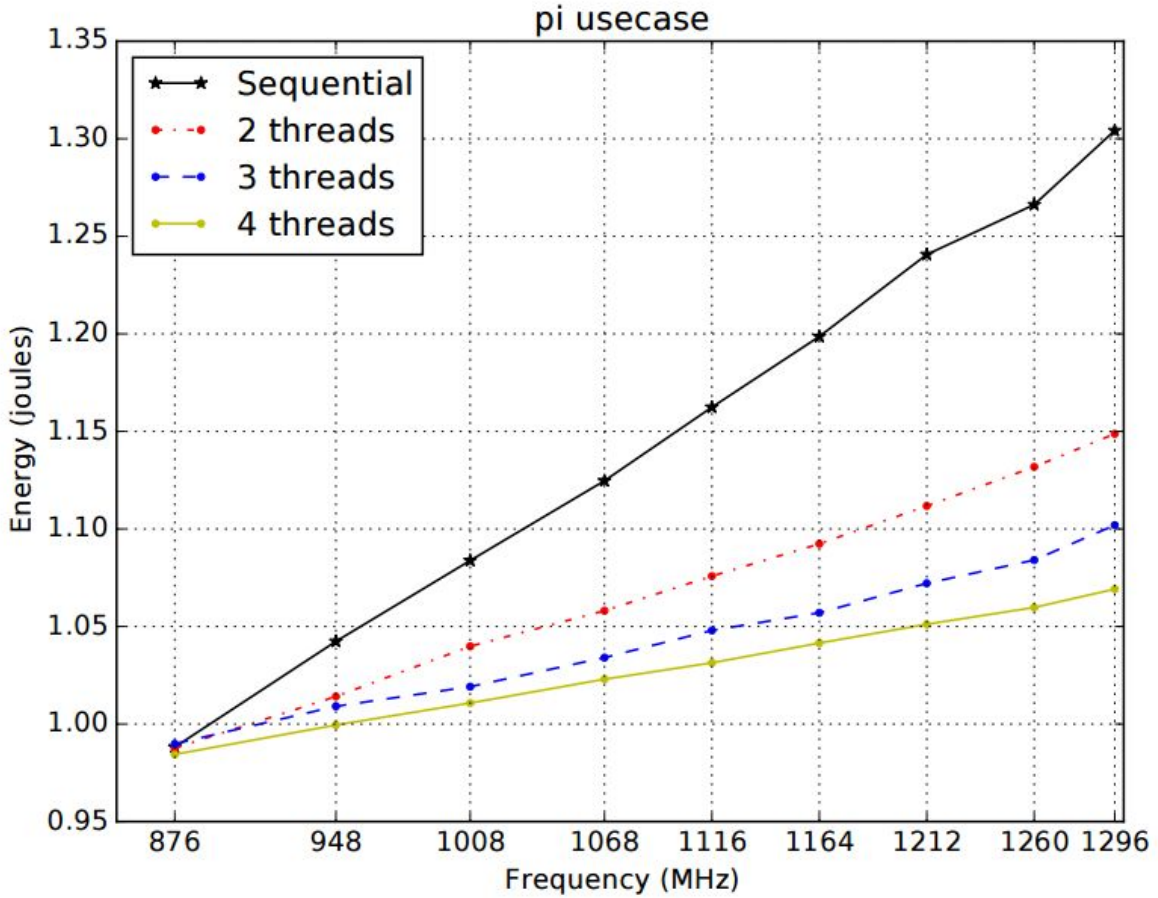




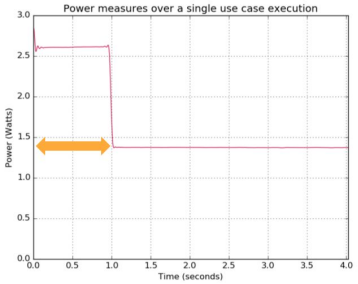
$$P \propto k f^3$$

Use case alone - energy

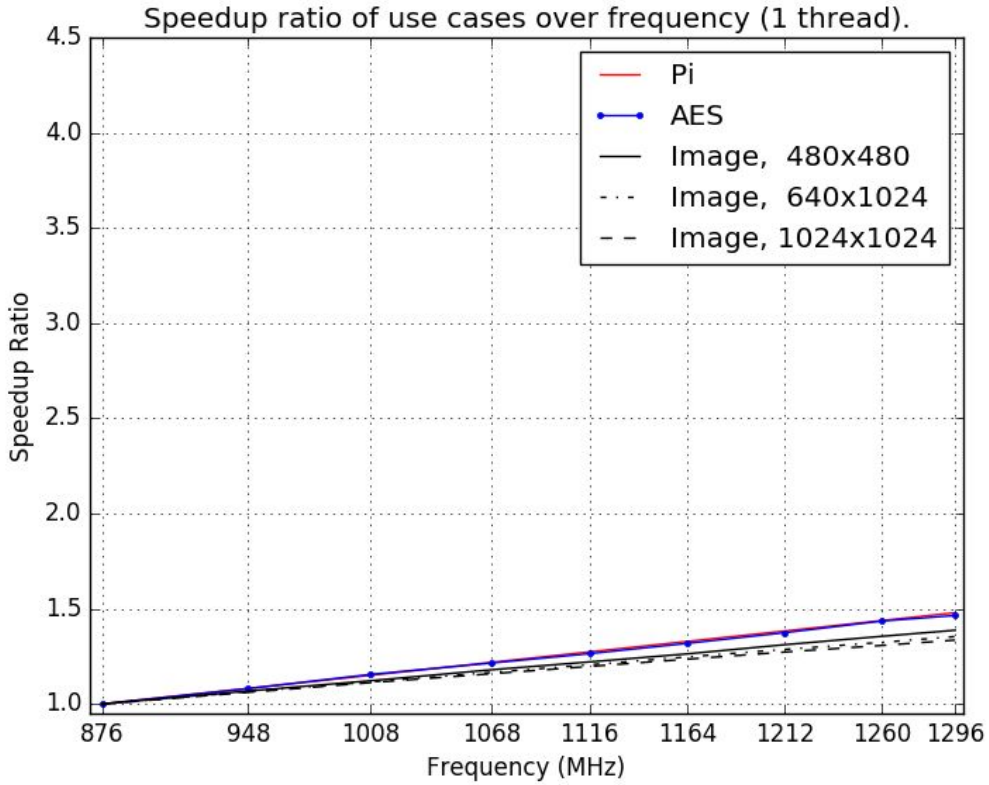




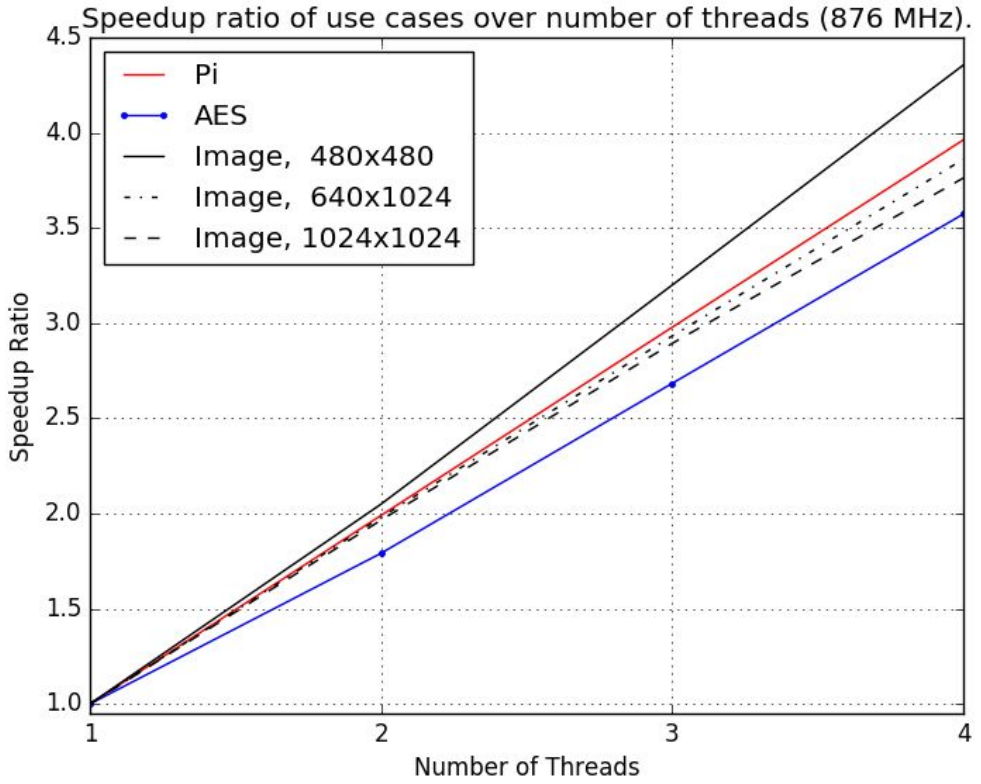
Execution time speedup



Frequency scaling



OpenMP parallelism



1. Run-time framework
2. Task model and analysis
3. Experiments
    a. Testbed description
    b. Single use cases
    c. Task systems

System experiments

System experiments

- Generate 232 random task systems (U = 0.6 → 2.9)

System experiments

- Generate 232 random task systems (U = 0.6 → 2.9)
- Bind generated tasks to use cases

System experiments

- Generate 232 random task systems (U = 0.6 → 2.9)
- Bind generated tasks to use cases
- Scheduling test & optimisation in Python for **1, 2, 3, 4 threads**

System experiments

- Generate 232 random task systems (U = 0.6 → 2.9)
- Bind generated tasks to use cases
- Scheduling test & optimisation in Python for **1, 2, 3, 4 threads**
- Operating points and threads partition for each task

System experiments

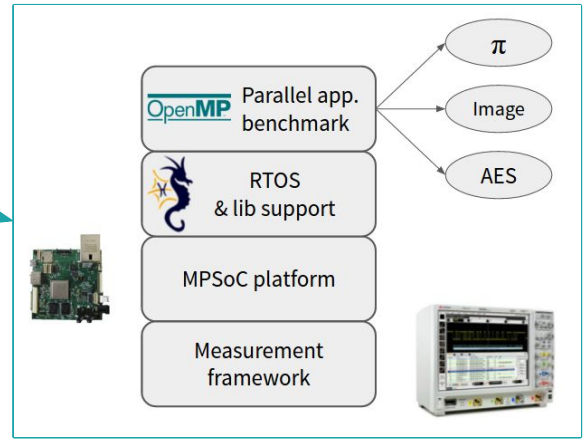
- Generate 232 random task systems (U = 0.6 → 2.9)
- Bind generated tasks to use cases
- Scheduling test & optimisation in Python for **1, 2, 3, 4 threads**
- Operating points and threads partition for each task
- Generate HIPPEROS builds

System experiments

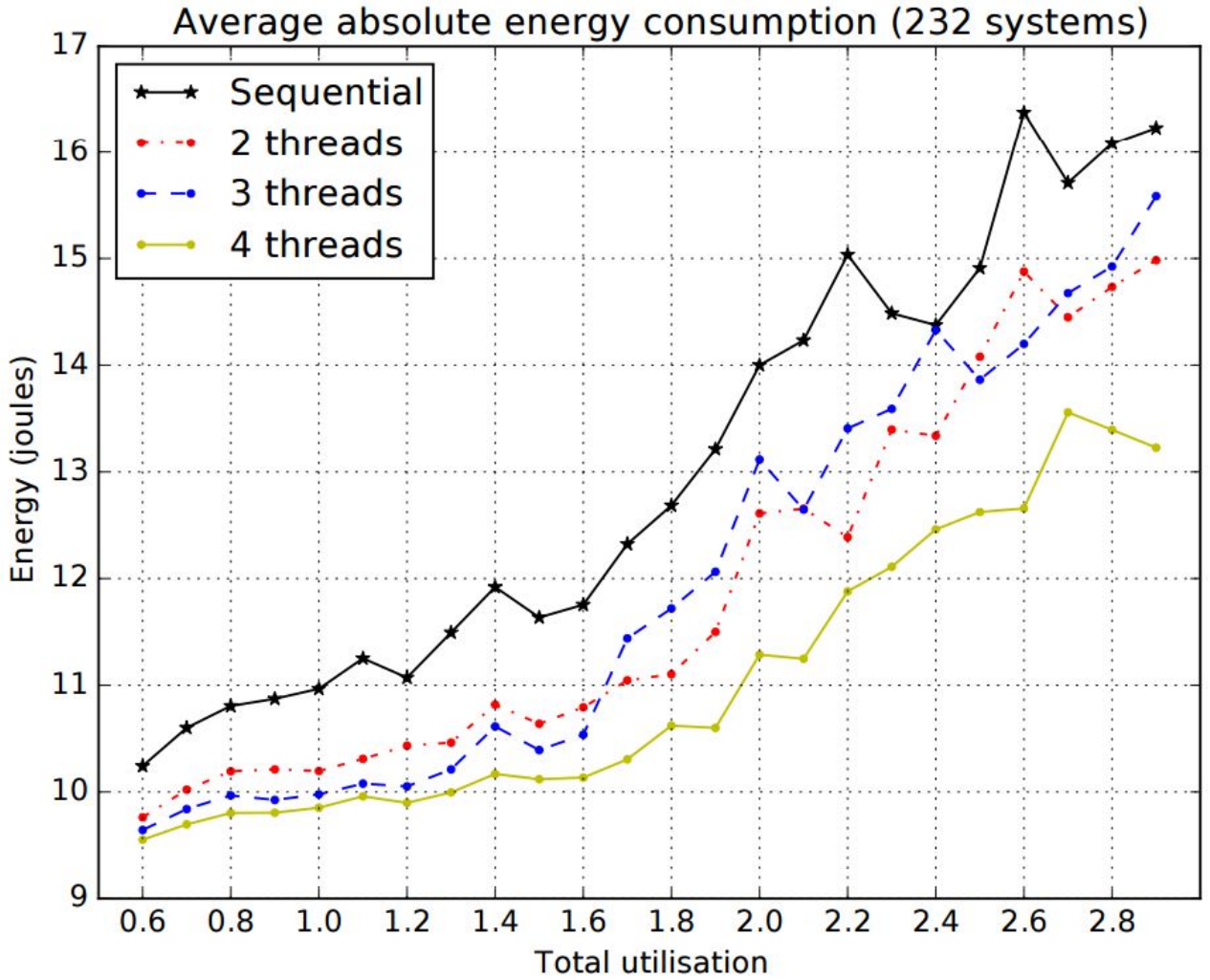
- Generate 232 random task systems (U = 0.6 → 2.9)
- Bind generated tasks to use cases
- Scheduling test & optimisation in Python for **1, 2, 3, 4 threads**
- Operating points and threads partition for each task
- Generate HIPPEROS builds
- Run feasible systems and measures

System experiments

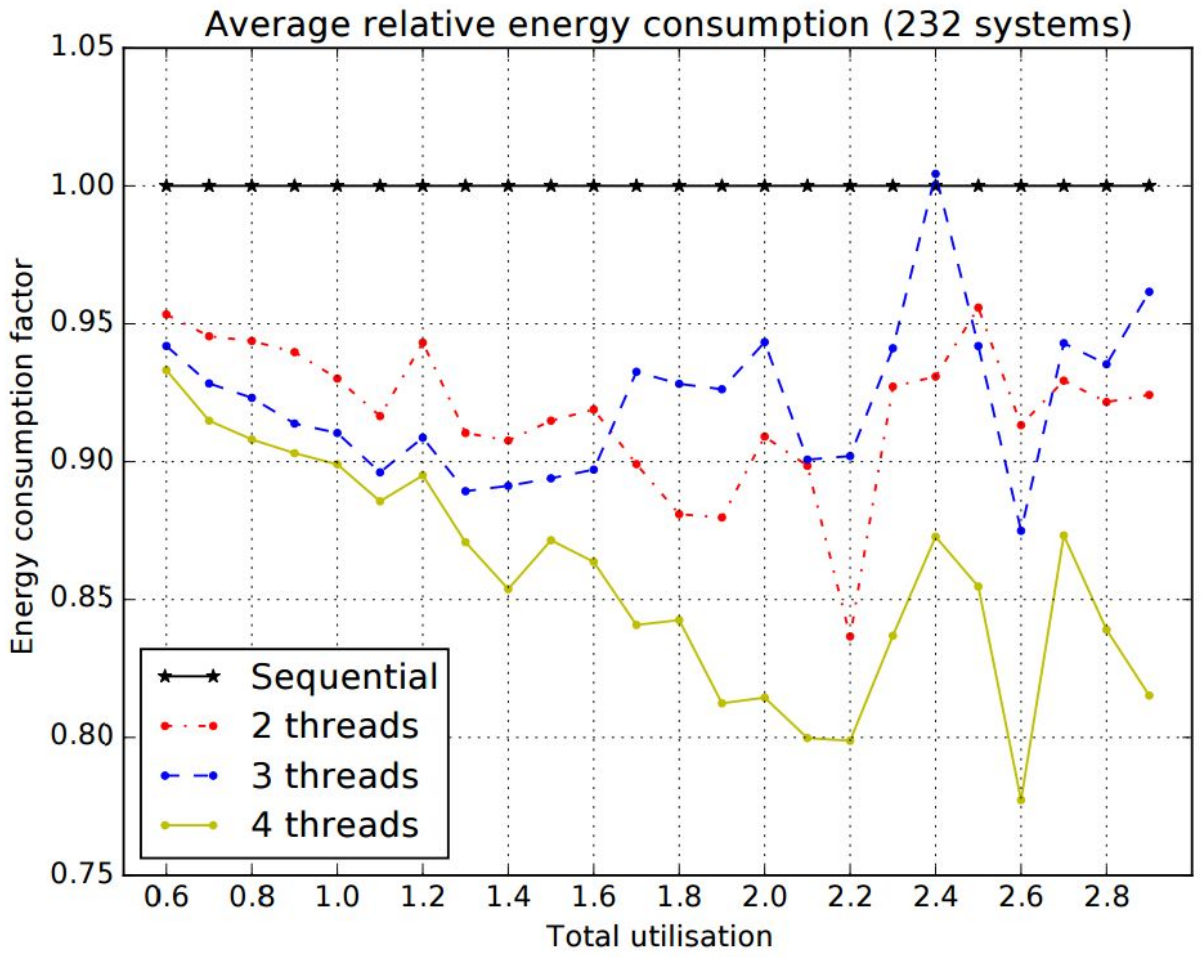
- Generate 232 random task systems (U = 0.6 → 2.9)
- Bind generated tasks to use cases
- Scheduling test & optimisation in Python for **1, 2, 3, 4 threads**
- Operating points and threads partition for each task
- Generate HIPPEROS builds
- Run feasible systems and measures



Energy consumption



Relative energy savings



Conclusions

- Practical experimental framework flow for parallel real-time applications
- Parallelisation saves up to 25% energy (the whole board)
- Confronted theory with practice
  - Speedup factors
  - Power measurements
- Challenge: integrate “OpenMP-like” programs in industry systems

Conclusion

Parallelism helps to **reduce energy**
while meeting **real-time requirements**

Thank you.

Questions?

Dynamic power vs static leakage

$$P \propto k\ f^3 + \alpha\ k$$

Dynamic power vs static leakage

$$P \propto \underbrace{k\ f^3}_{\text{dynamic}} + \underbrace{\alpha\ k}_{\text{static}}$$

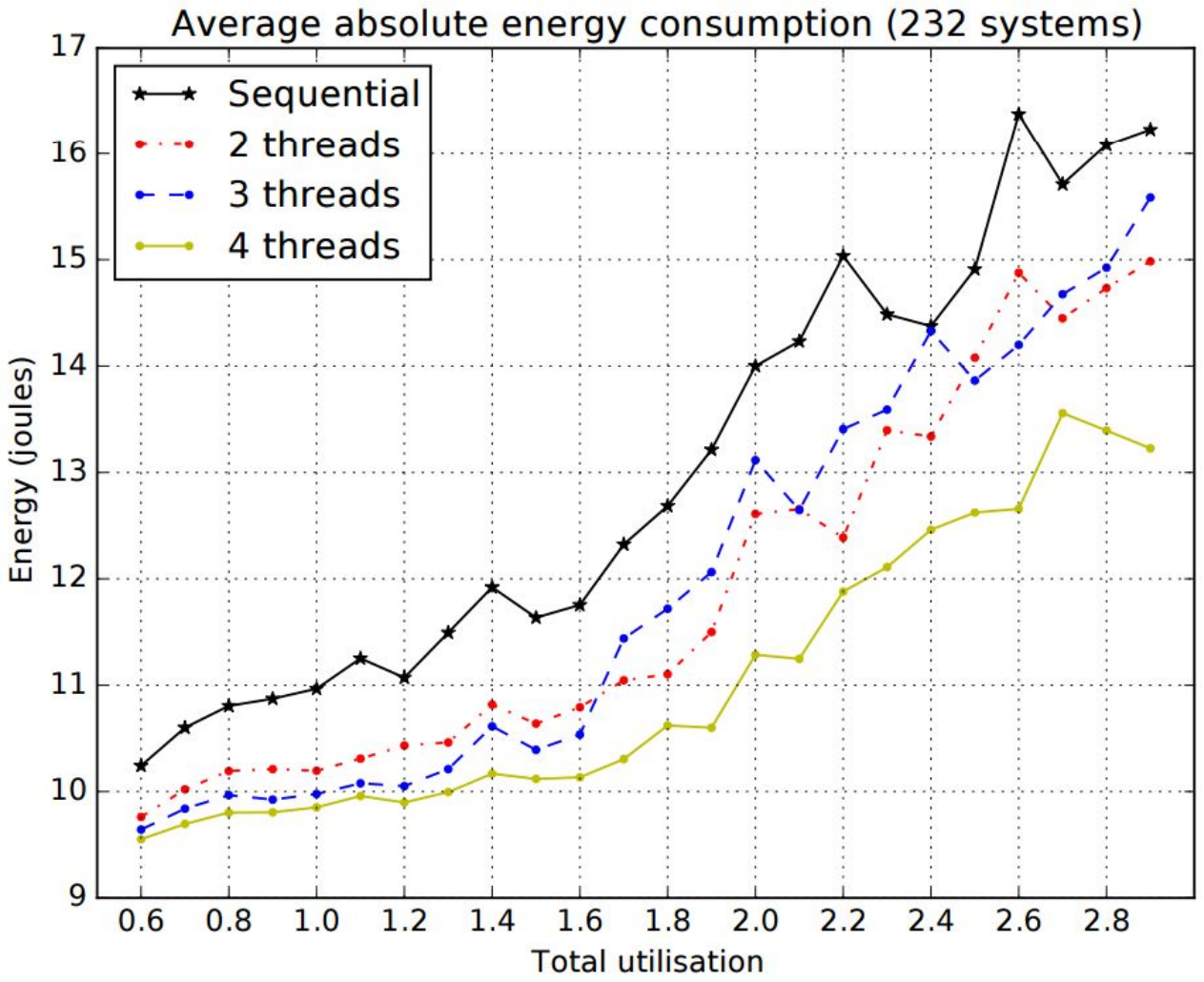
Dynamic power vs static leakage

$$P \propto \underbrace{k\ f^3}_{\text{dynamic}} + \underbrace{\alpha\ k}_{\text{static}}$$

- α is platform-dependent
- Comparison between DVFS and DPM

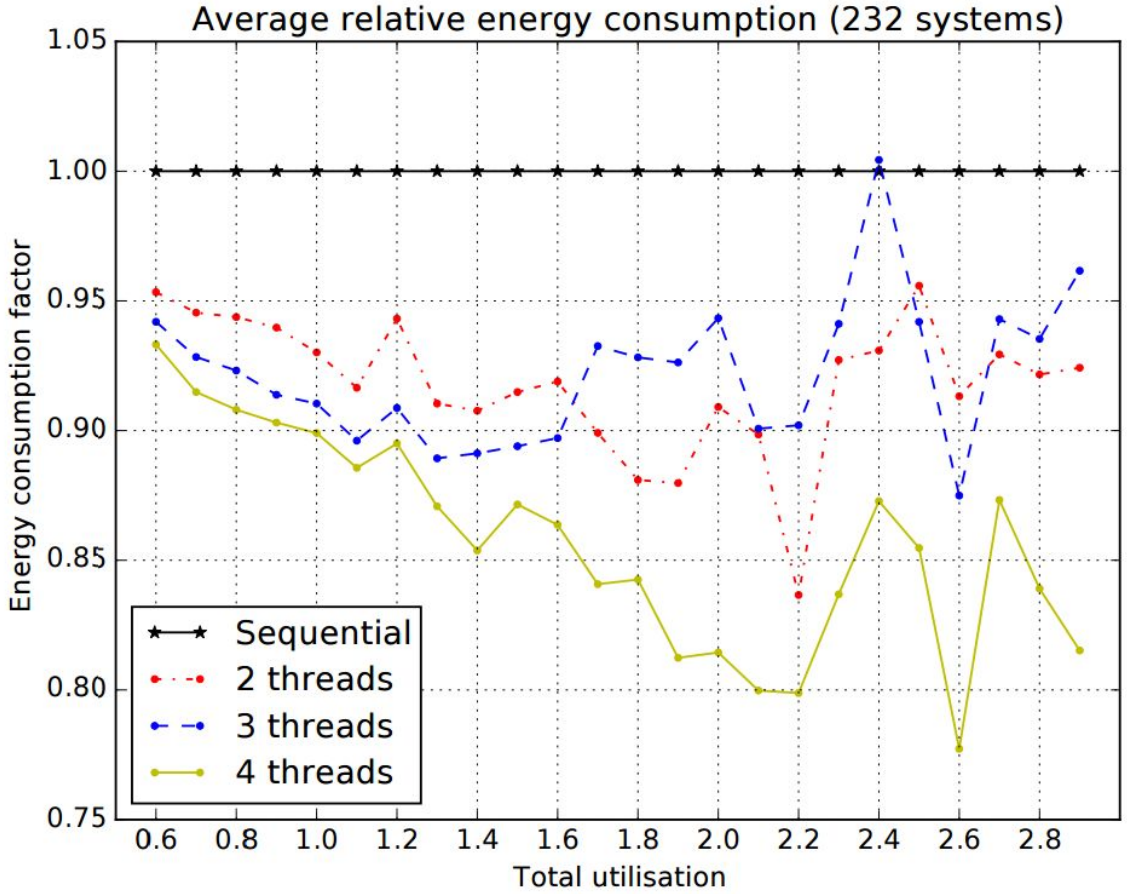
Energy consumption

- 232 systems for each degree of //
  - No high utilisation systems
    → Partitioned Rate Monotonic
  - Only feasible systems for 1 → 4 threads
- More threads consume less
- Low utilisation systems don't need high operating points (often *idle*)
- Analysis is pessimistic for high number of threads



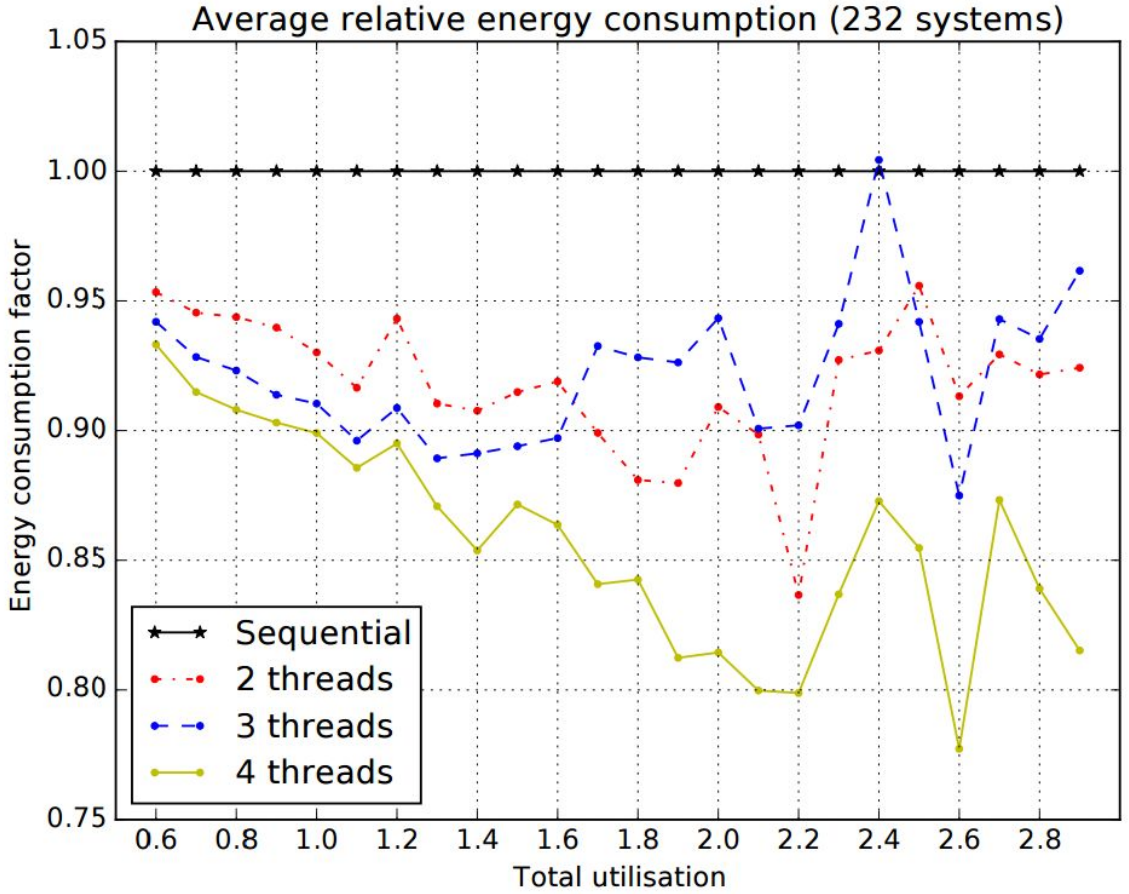
Relative energy savings

- Same systems, but rel. to 1 thread
- **Parallelism helps to save energy :-)**
- $\sigma < 0.13$, but not a lot of systems...
- "4 threads" dominates
- $\nearrow U_{tot} \Rightarrow \searrow$ energy
  - Until $U_{tot} = 2.4$
  - For high U_{tot} :
    - Analysis pessimism
    - Schedulable systems only bias, so already well distributed
    - Some systems are only schedulable in parallel



Questions on experiment results

- “3 threads” is bad:
  - Loss of symmetry with 4 cores
  - Different tasks executes simultaneously (1+3)
- Measures on the whole platform
  - CPU alone would give better results
- Need more data
  - Charts would be smoother
  - But it takes time…
    - 1 minute/execution
    - 4 executions/system
    - 232 systems, ≈ 15 hours



Analysis technique

- Flawed, but does not impact results
  - Axer *et al* (ECRTS'13)
  - Flaw pointed by Fonseca *et al* (SIES'16)
- Scheduling framework
  - Part of the *technical* framework
  - Not the important contribution/result
- Will be improved with future work