Trace-driven Simulation of Multithreaded Applications

Alejandro Rico, Alejandro Duran, Felipe Cabarcas
Yoav Etsion, Alex Ramirez and Mateo Valero

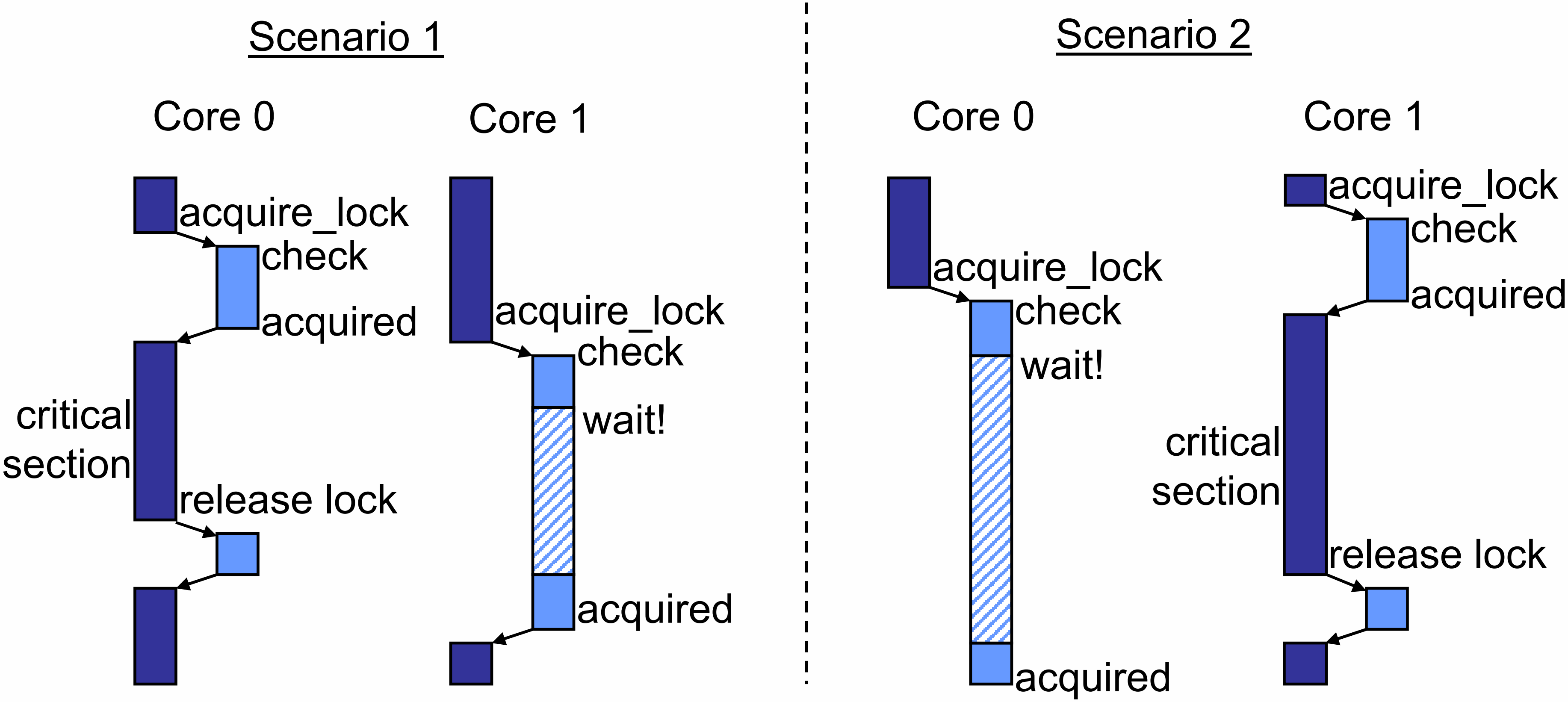
UNIVERSITAT POLITÈCNICA DE CATALUNYA

UNIVERSIDAD DE ANTIOQUIA

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Multithreaded applications and trace-driven simulation

- Most computer architecture research employ execution-driven simulation tools.
- Trace-driven simulation cannot capture the dynamic behavior of multithreaded applications.



Trace-driven simulation has advantages

- Avoid computational requirements of simulated applications.
  - Memory footprint.
  - Disk space for input sets.
- Simulate applications with non-accessible sources, but accessible traces.
  - Confidential/restricted applications.
- Lower modeling complexity.
  - Different host[1] and target[2] ISAs / endianness.

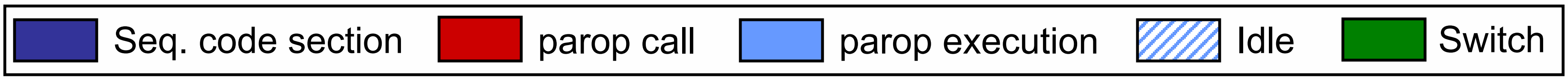
- **Problem: How to appropriately simulate multithreaded applications using traces?**

[1]*Host*: system where the simulator executes.

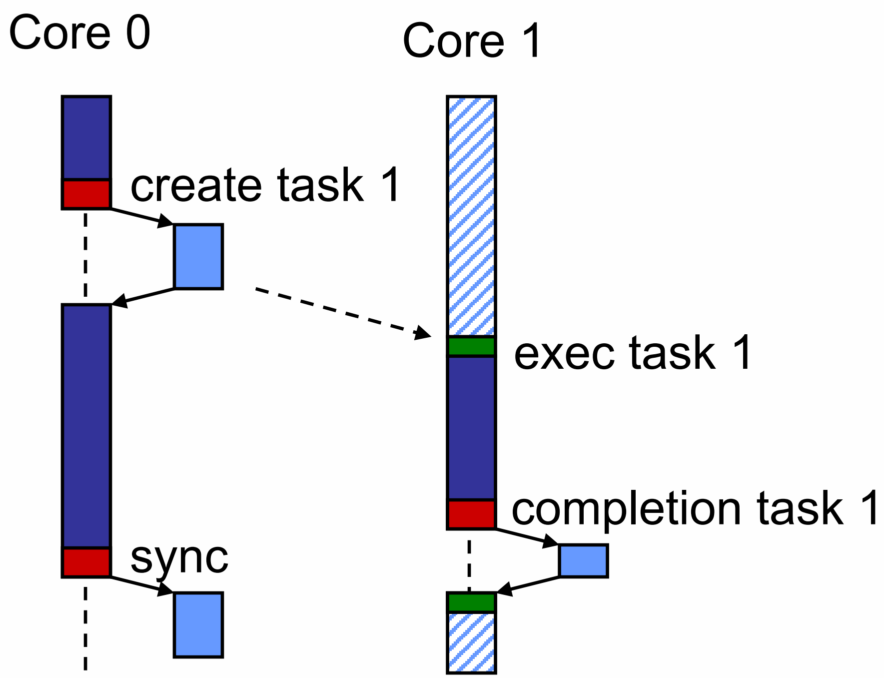
[2]*Target*: system modeled in the simulator.

Targeting applications with decoupled execution

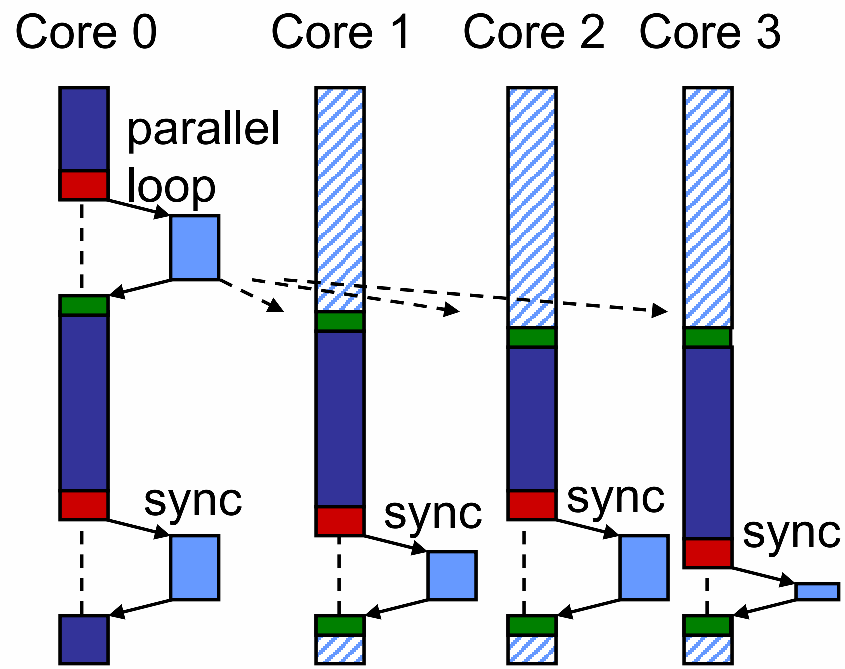
- Distinguish the user code (sequential code sections) from parallelism-management operations (*parops*).



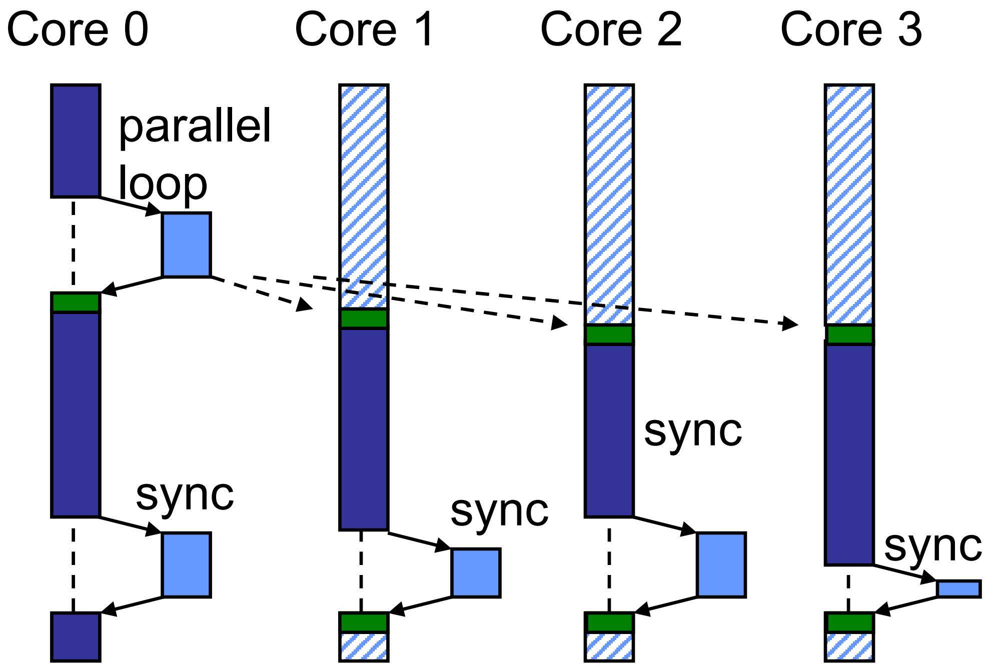
Task-based parallel applications



Loop-based parallel applications

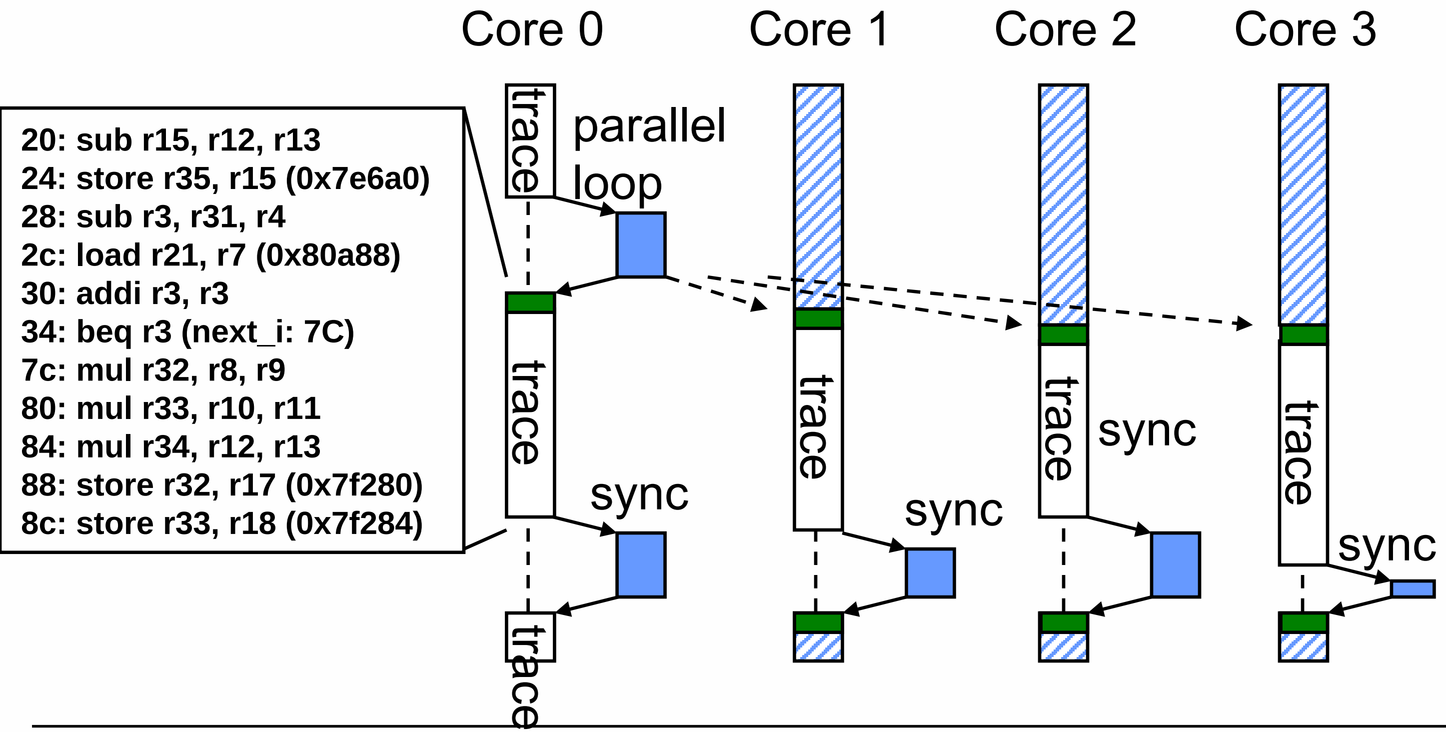


How traces are collected (I)



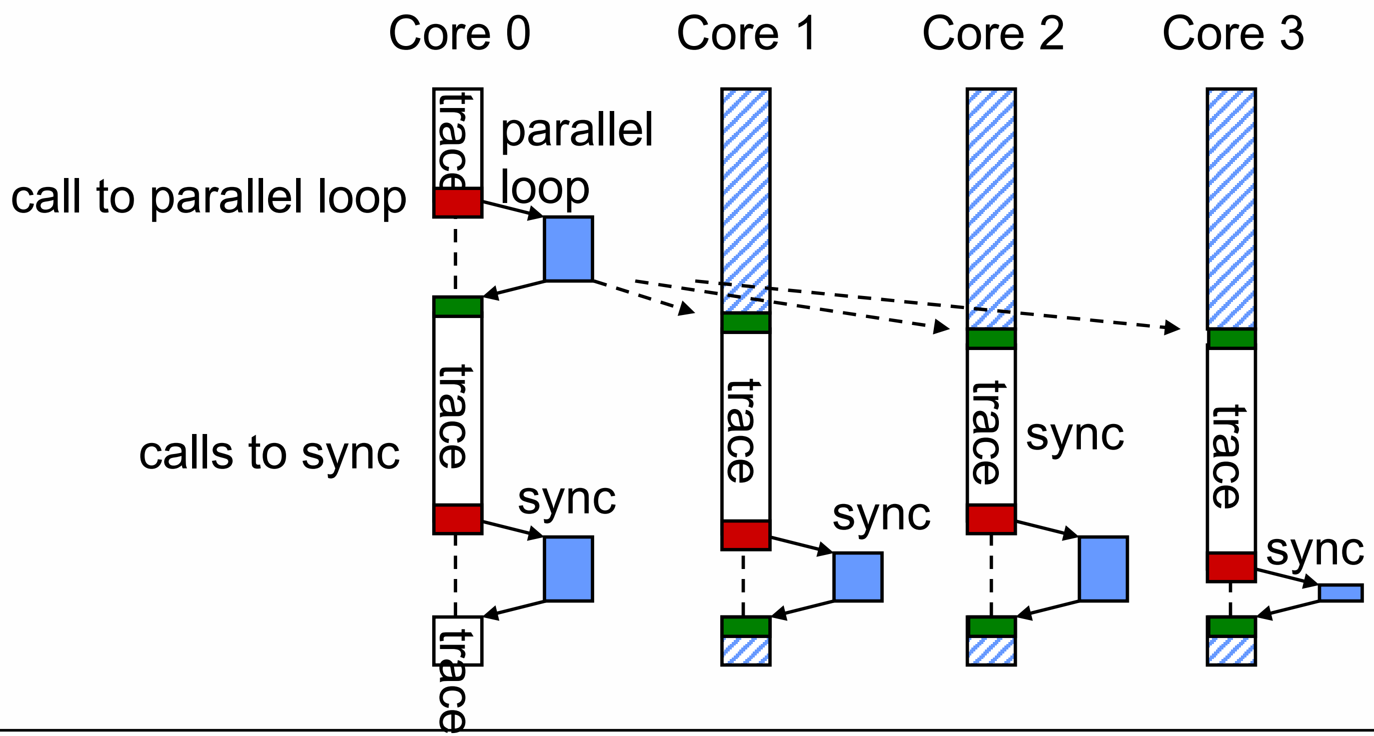
How traces are collected (II)

- Capture traces for sequential code sections. [trace]
  - Execution is independent of the environment.



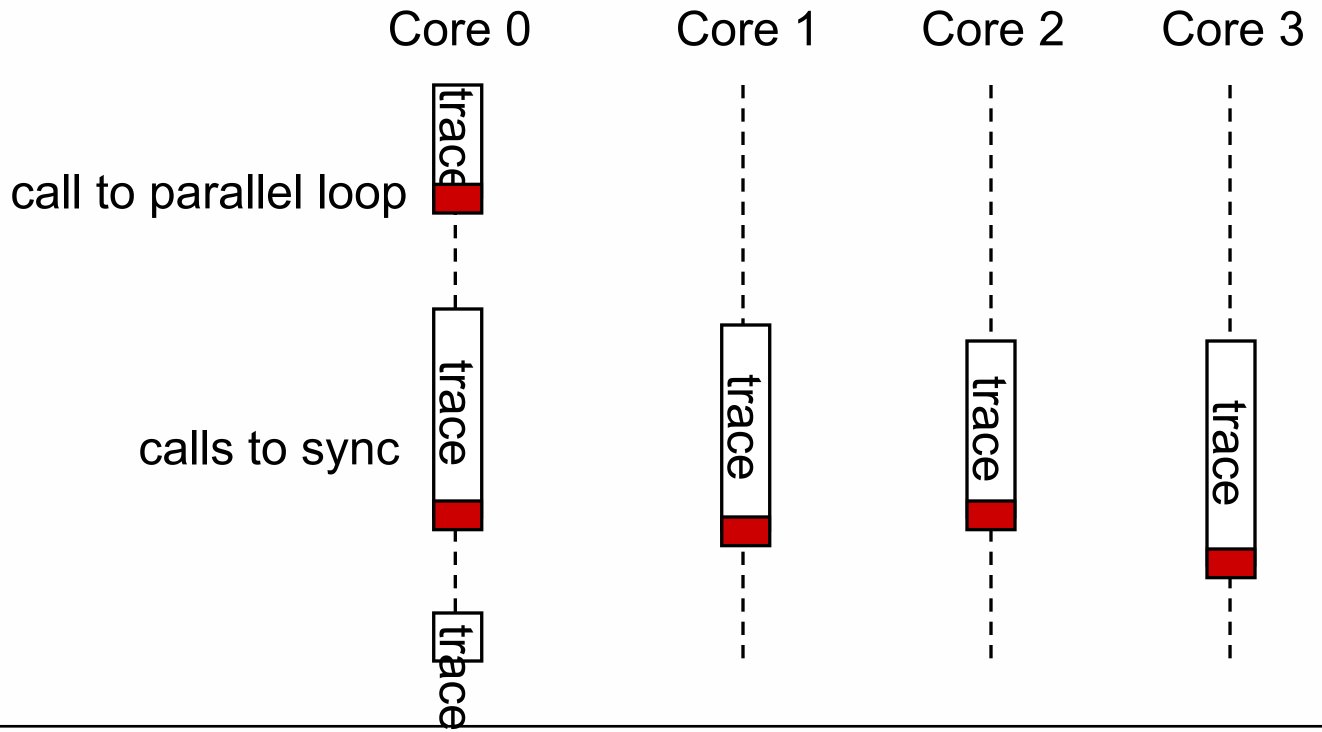
How traces are collected (III)

- Capture traces for sequential code sections. trace
  - Execution is independent of the environment.
- Capture <u>calls</u> to *parops*.
  - Specific *parop call* events are included in the trace.



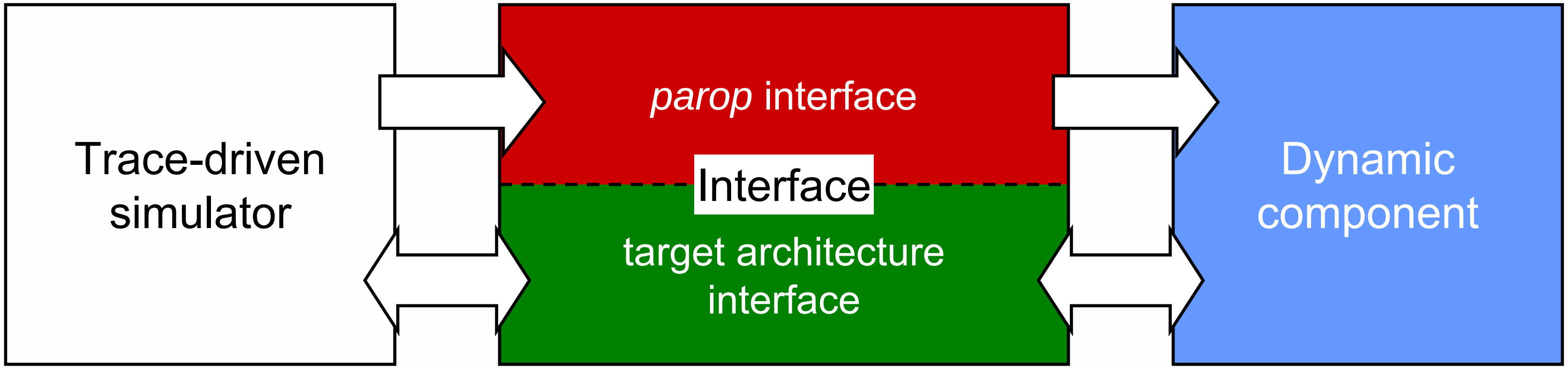
How traces are collected (IV)

- Capture traces for sequential code sections. [trace]
  - Execution is independent of the environment.
- Capture <u>calls</u> to *parops*.
  - Specific *parop call* events are included in the trace.
- Do <u>not</u> capture the execution of *parops*.
  - Execution depends on the environment.



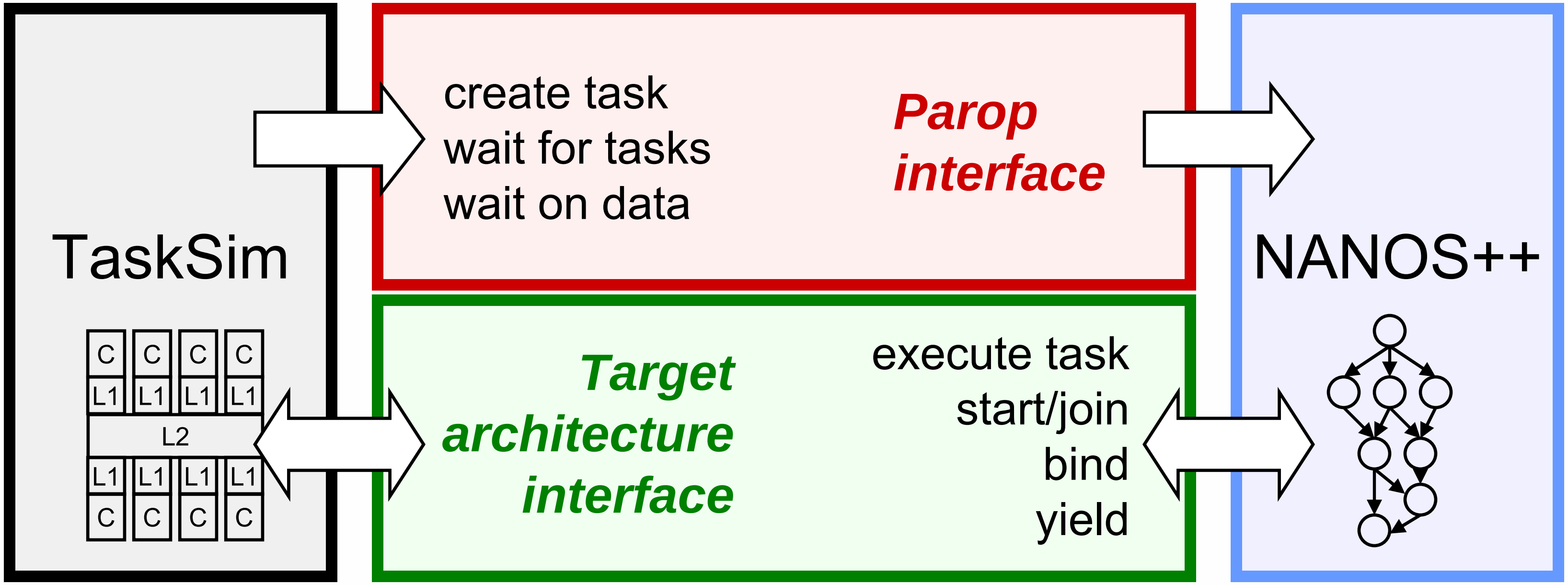
Simulation framework

- Trace-driven simulator simulates *sequential code sections*.
- The dynamic component executes parops at simulation time.
  - Includes the implementation of parops.
- Parops are exposed to the simulator through the parop interface.
- The architecture state is exposed to the dynamic component through the target architecture interface.



Sample implementation: TaskSim – NANOS++

- *Parops* are exposed to the simulator through the *parop interface*
  - It includes operations for task management and synchronization.
- The architecture state and associated actions are exposed to NANOS++ through the *architecture-dependent module.*
  - NANOS++ can alter the simulator state and manage the simulated thread according to the decisions based on the target architecture.



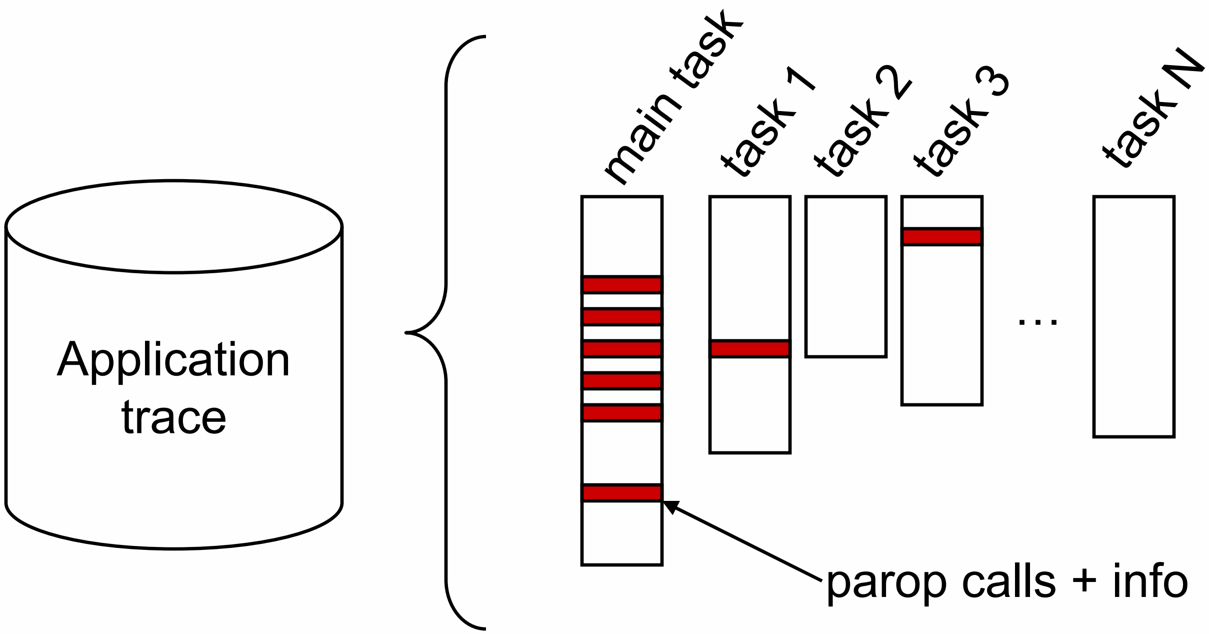
OmpSs application example

```
float A[N][N][M][M]; // NxN blocked matrix,
                     // with MxM blocks
for (int j = 0; j<N; j++) {
   for (int k = 0; k<j; k++)
      for (int i = j+1; i<N; i++)
         #pragma task input(a, b) inout(c)
         sgemm_t(A[i][k], A[j][k], A[i][j]);

   for (int i = 0; i<j; i++)
      #pragma task input(a) inout(b)
      ssyrk_t(A[j][i], A[j][j]);

   #pragma task inout(a)
   spotrf_t(A[j][j]);

   for (int i = j+1; i<N; i++)
      #pragma task input(a) inout(b)
      strsm_t(A[j][j], A[i][j]);
}
```

- Cholesky factorization.
- Tasks are spawned on *pragma task* annotations.
- Inputs and outputs are specified for automatic dependence resolution.

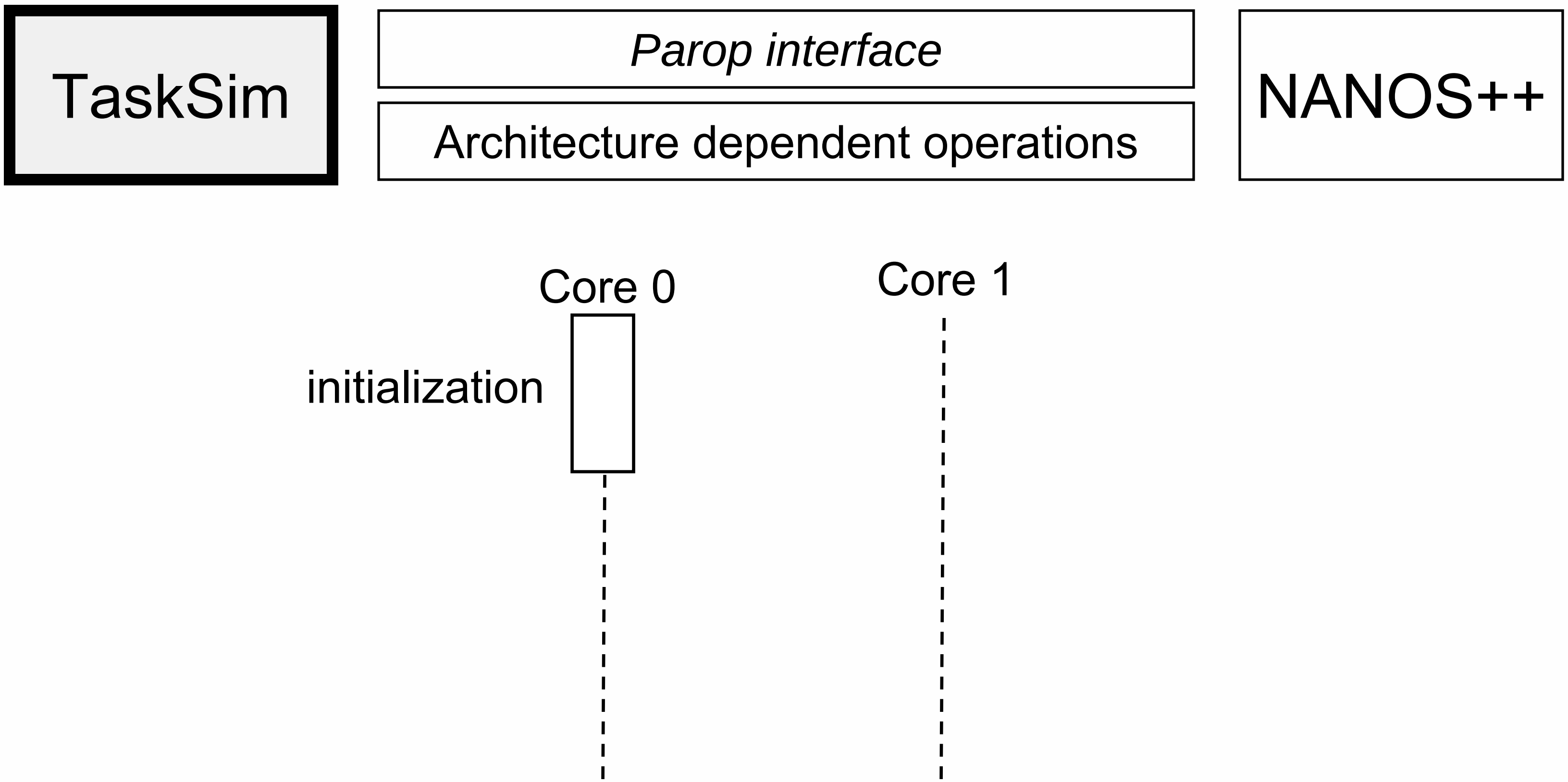
Traces for OmpSs applications

- Sequential code sections correspond to *tasks*.
- One trace for the main task
  - The thread starting the program execution at the *main* function
- One trace for each task
- Information for each function call
  - E.g., for task creation it needs the task id and the input and output data addresses and sizes



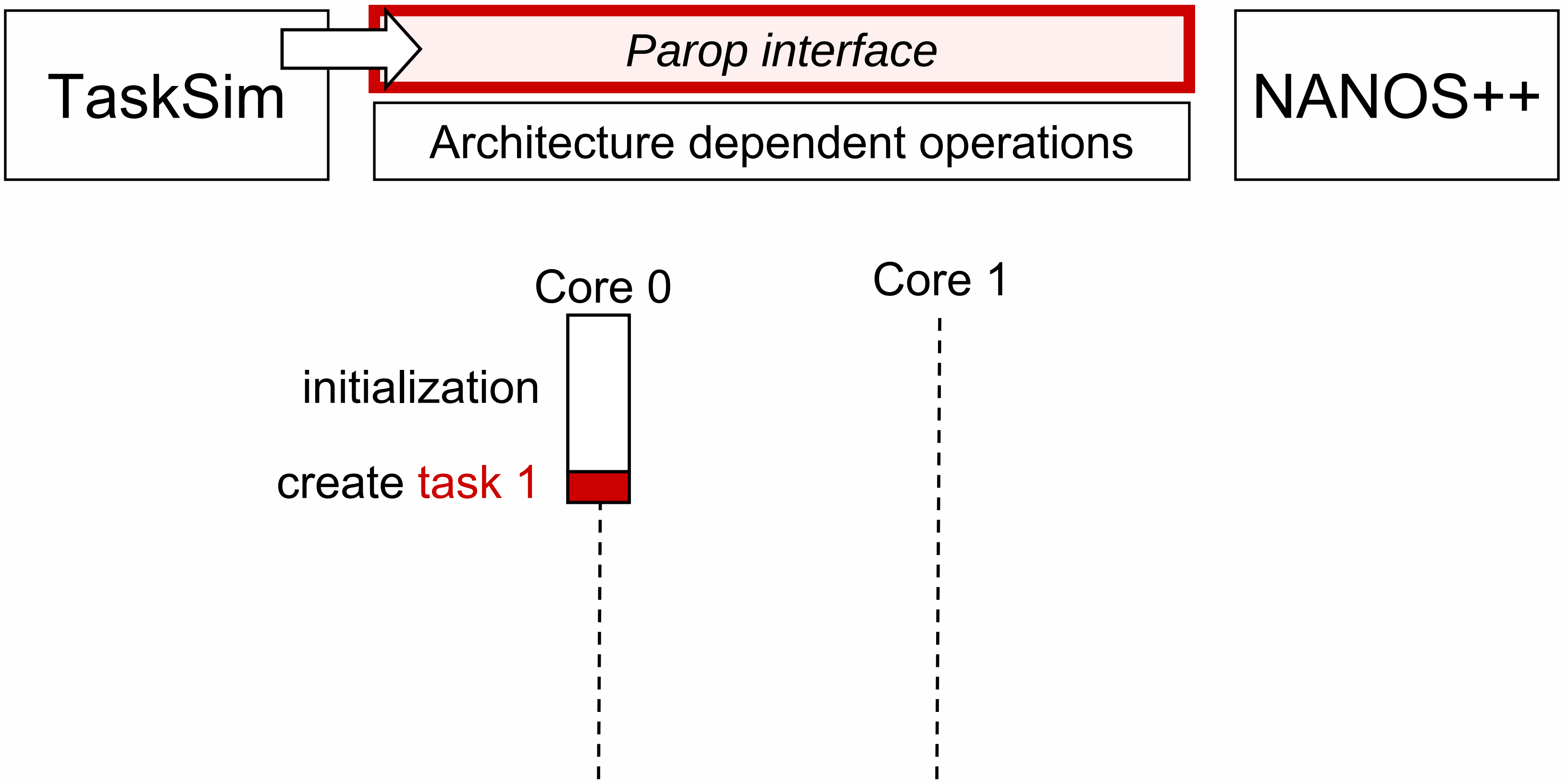
Simulation example (I)

1. Simulation starts the *main* task.



Simulation example (II)

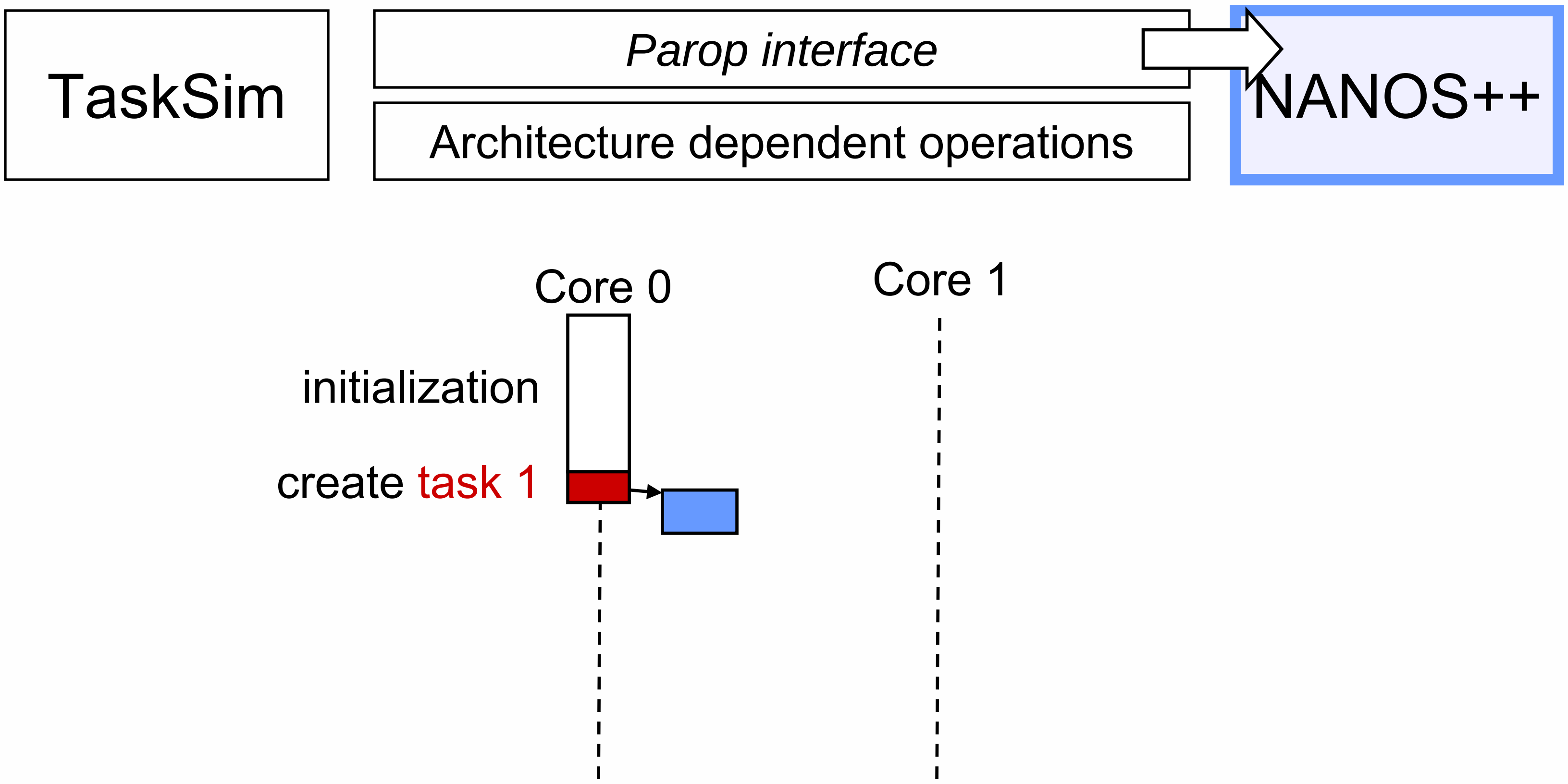
2. On a *create task* event, it calls the interface in the *Parop interface*.



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

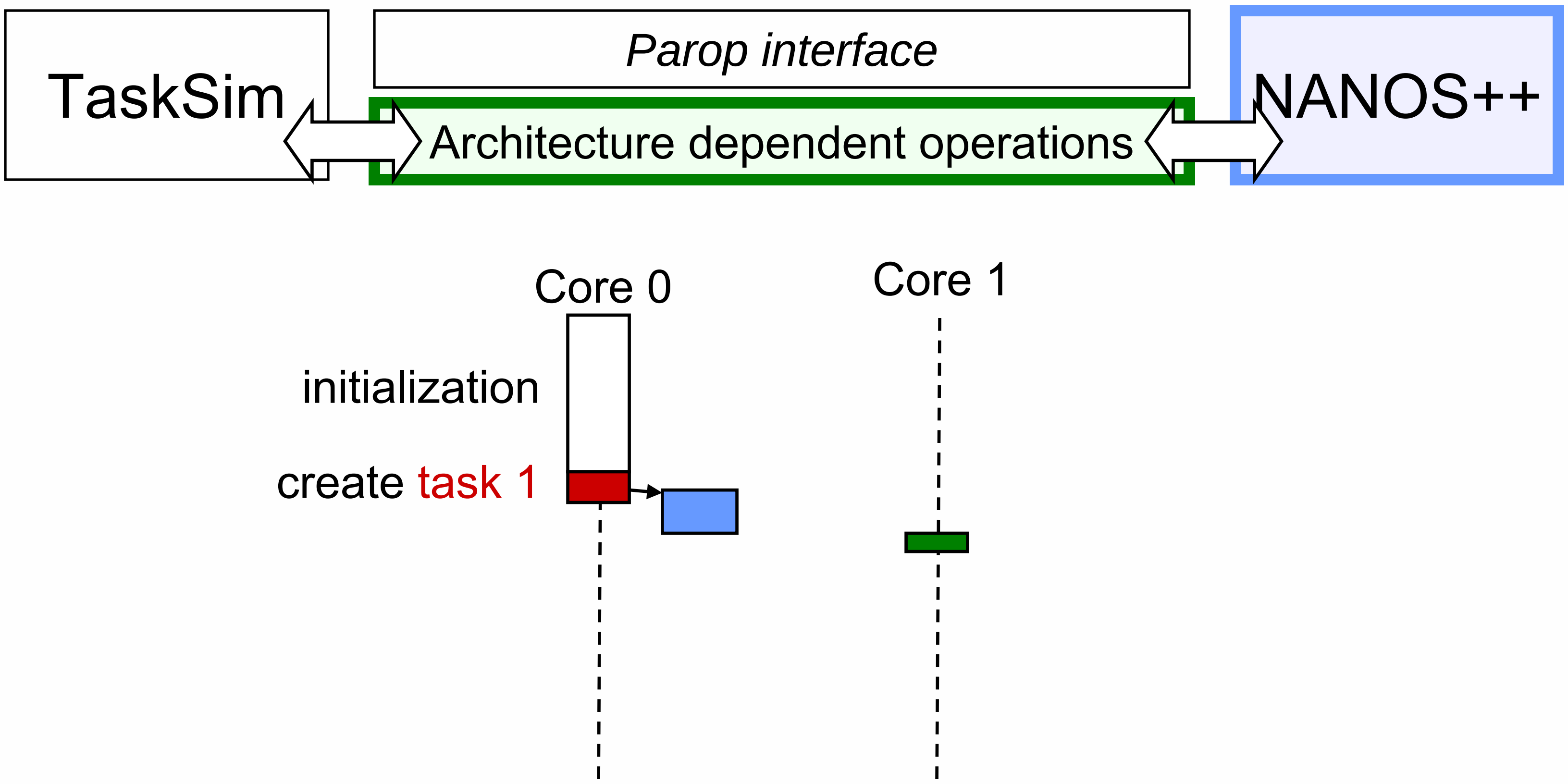
Simulation example (III)

3. That triggers the creation of the task in Nanos++.



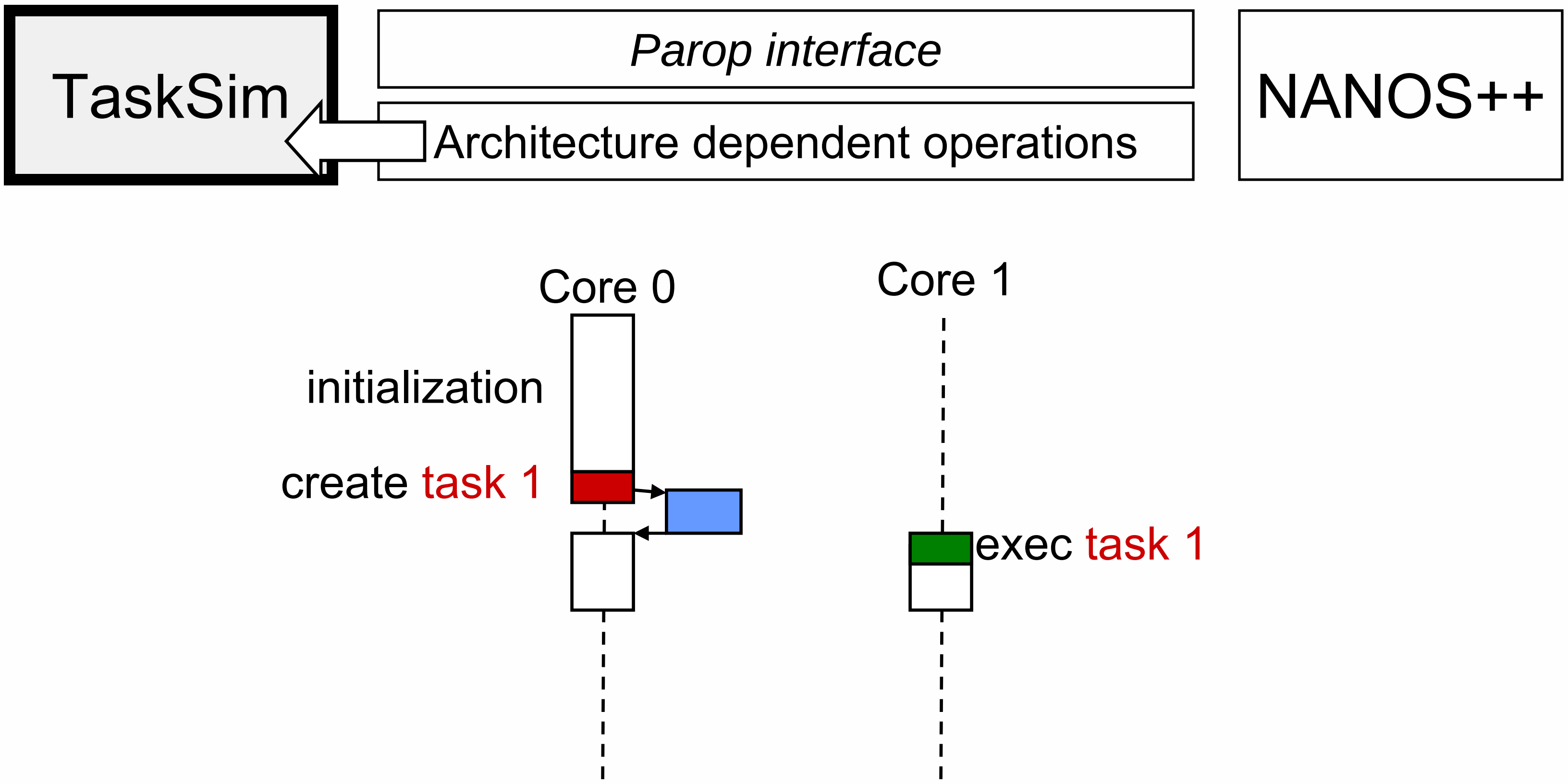
Simulation example (IV)

4. Returns control to TaskSim. Core 1 takes task 1 for simulation.



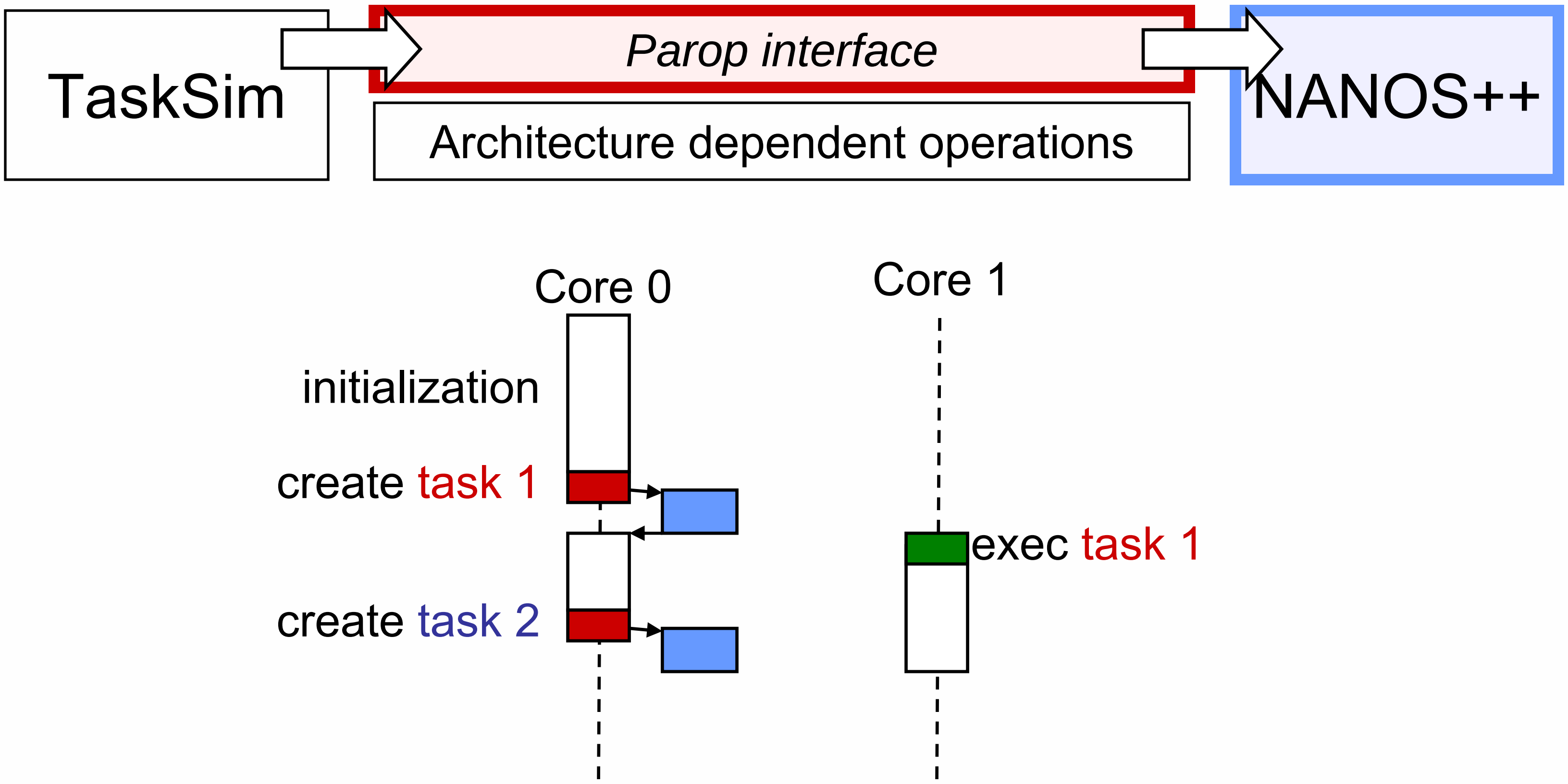
Simulation example (V)

5. TaskSim resumes simulation, and Core 1 starts simulating task 1.



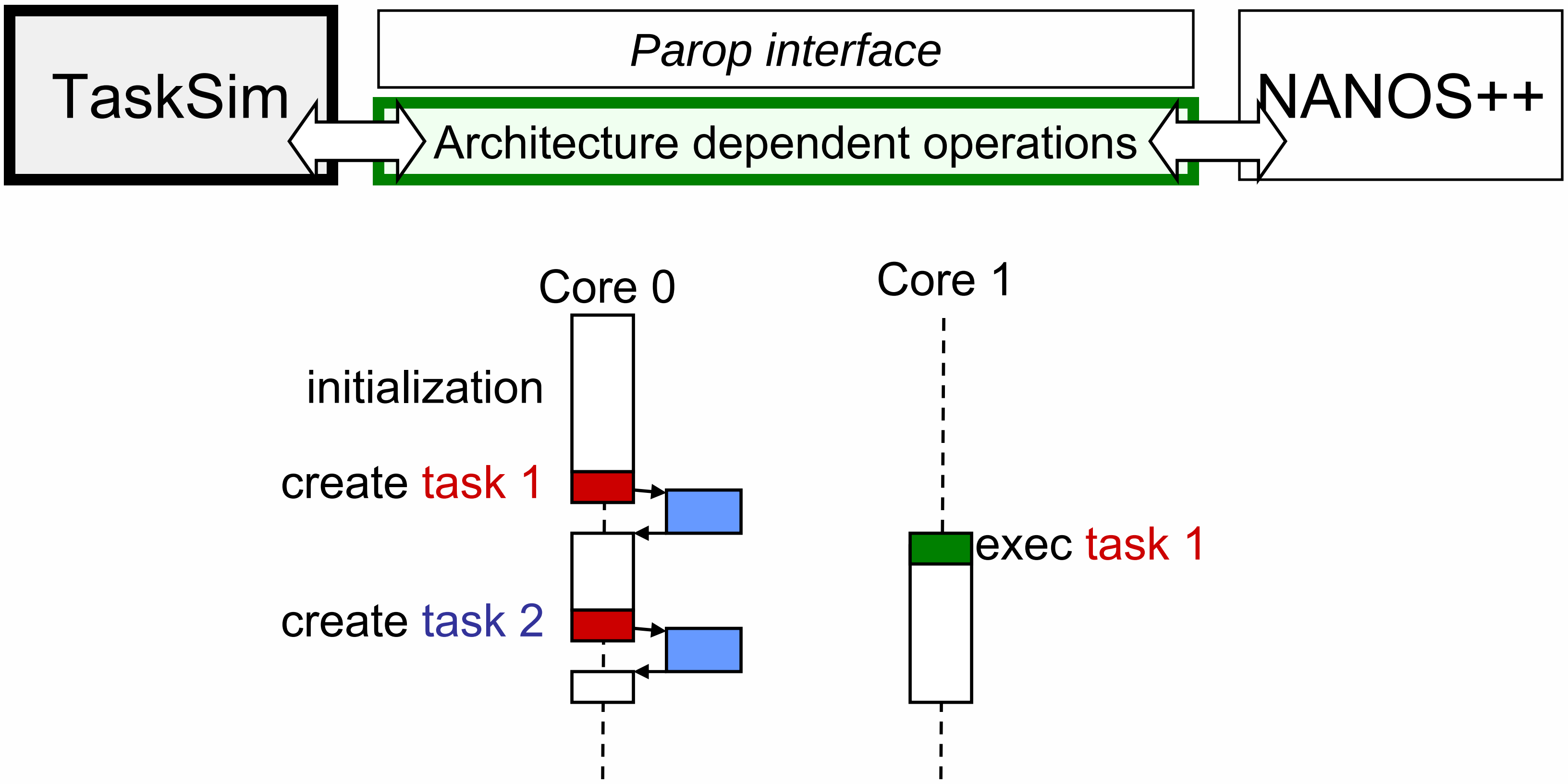
Simulation example (VI)

6. On create task 2 event, TaskSim calls the runtime again.

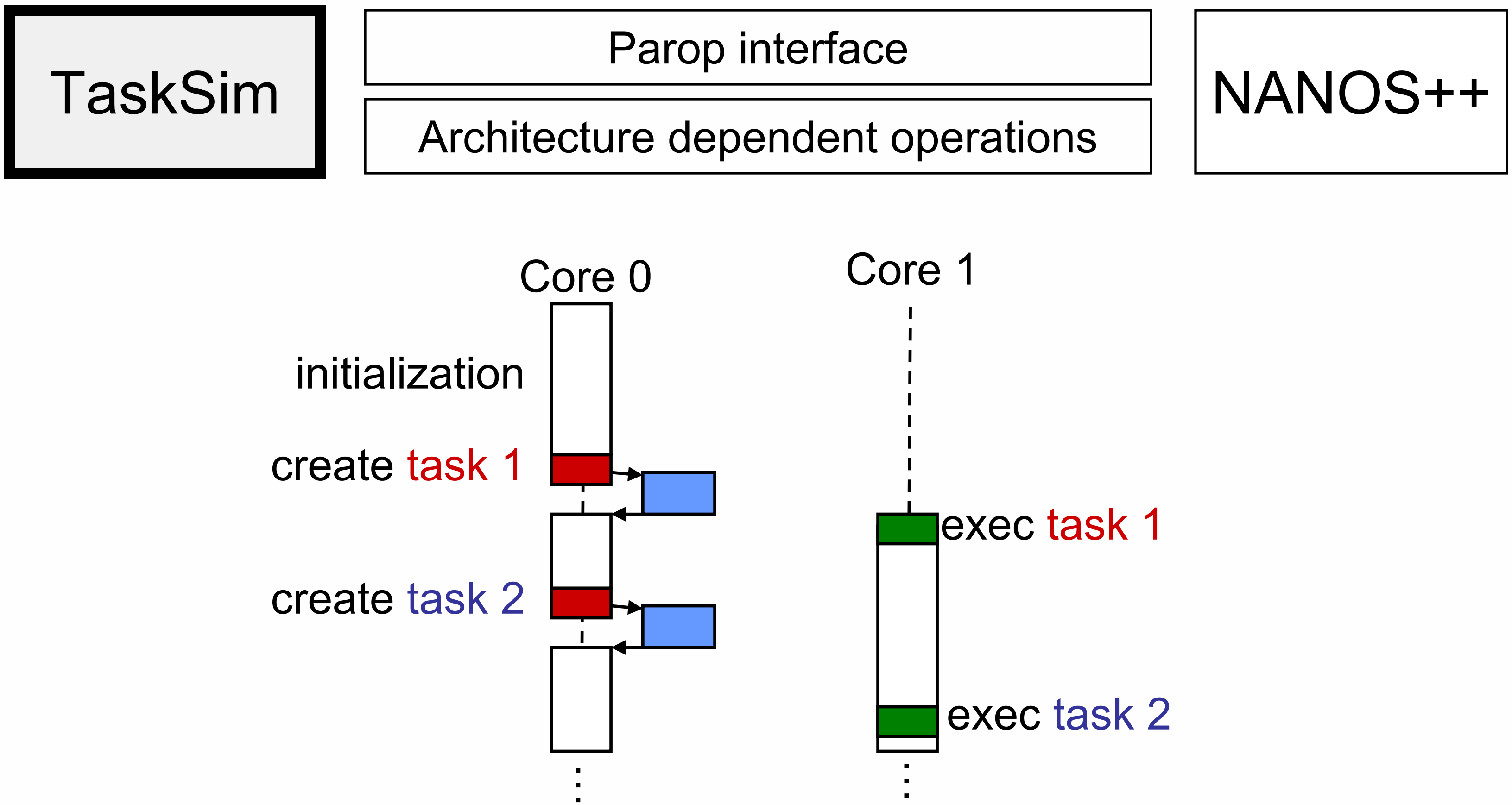


Simulation example (VII)

7. NANOS++ creates task 2, and returns control to TaskSim.

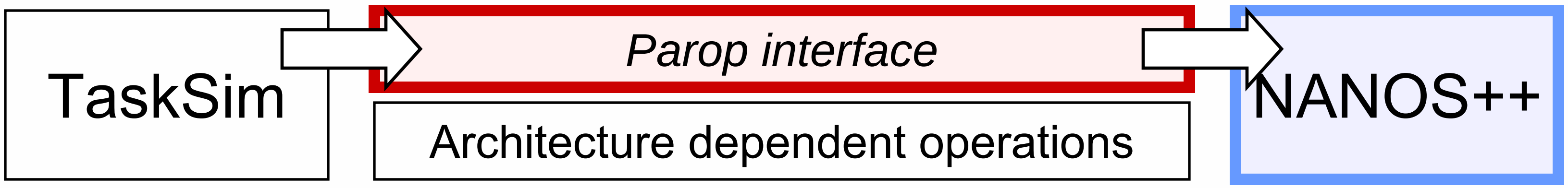


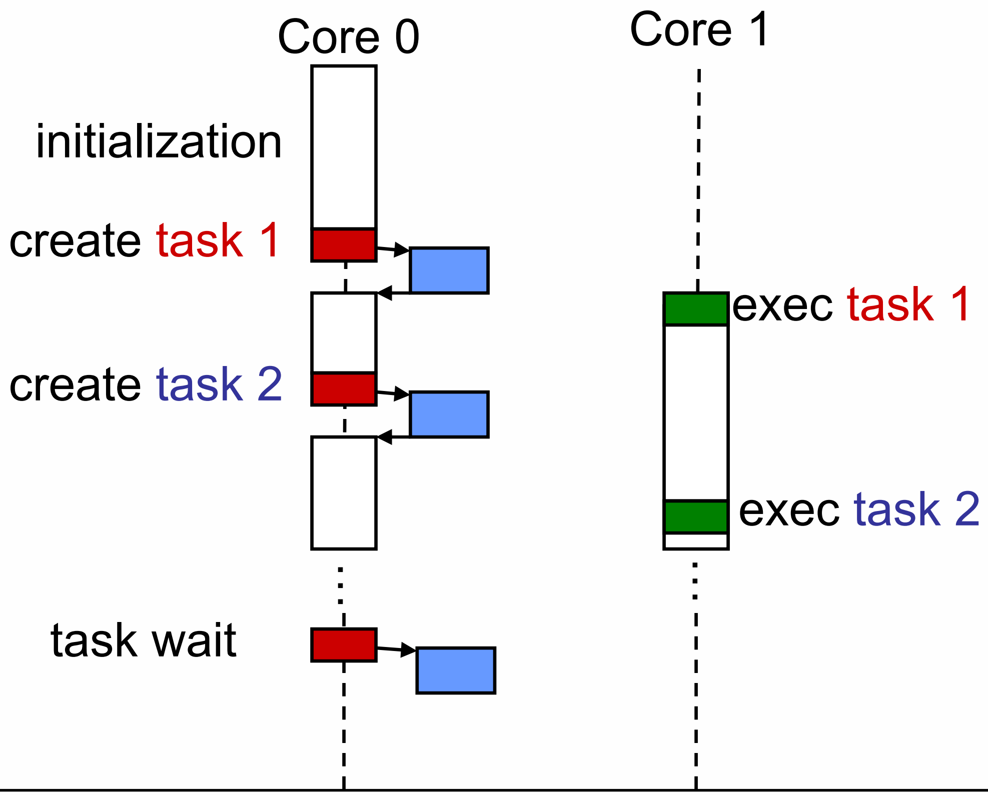
8. When Core 1 finishes the execution of task 1, starts task 2.



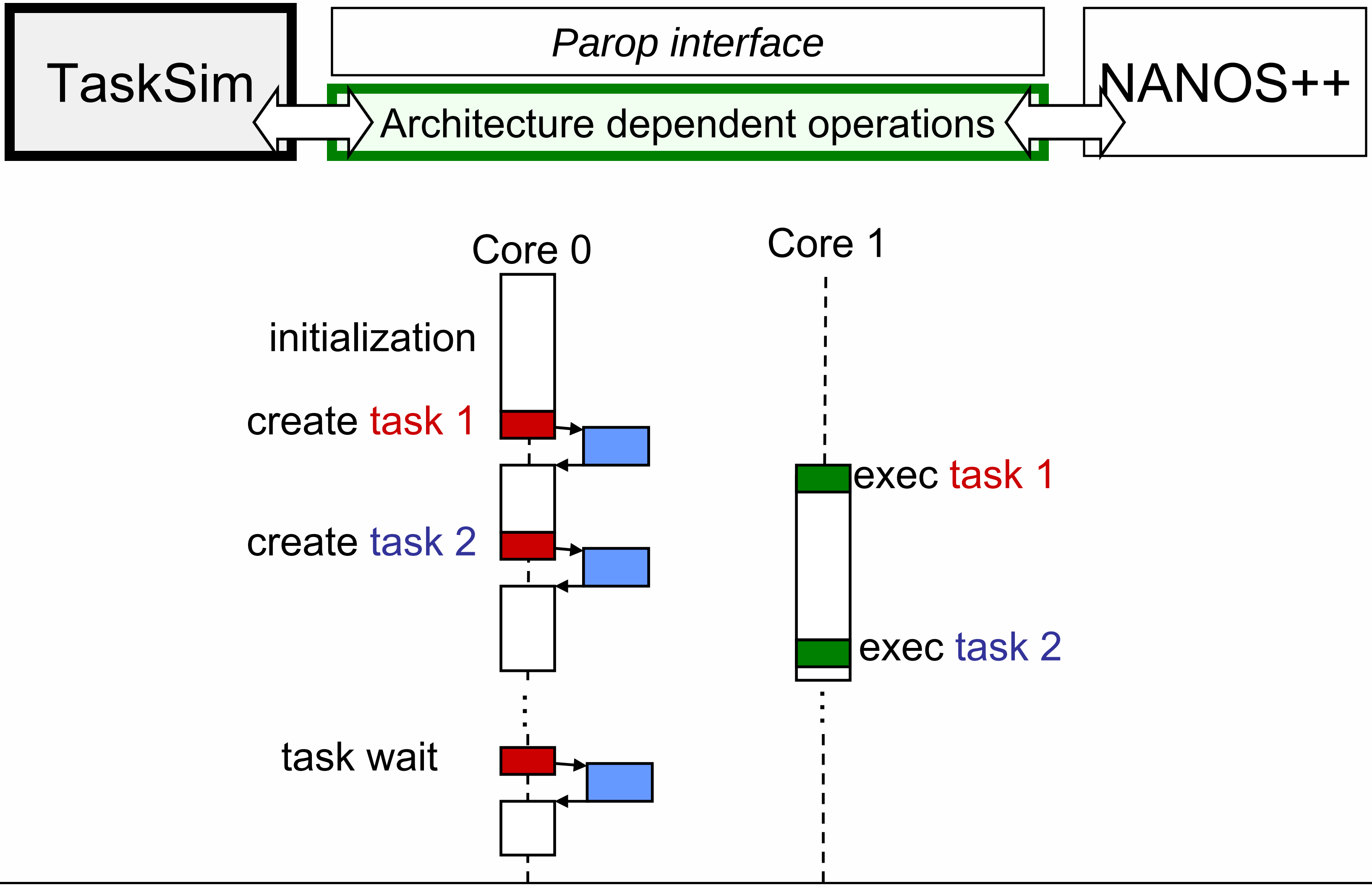
Simulation example (IX)

9. TaskSim reaches a synchronization *parop*. NANOS++ checks for pending tasks.

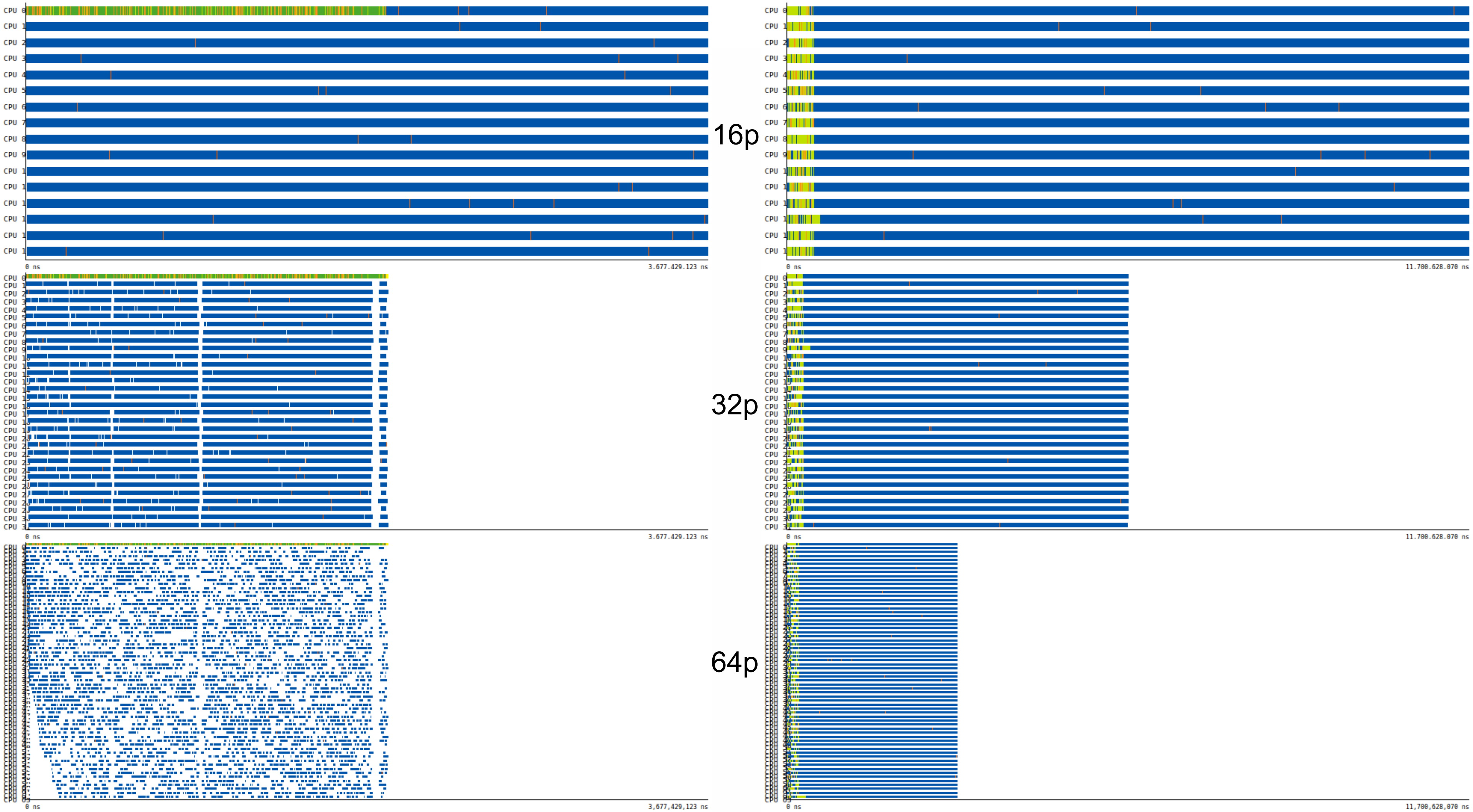




10. All tasks are finished, and TaskSim continues the main task simulation.



Task generation scheme scalability



- Task generation (green) on the *main task* limits scalability (on the left)
- Parallelization of task generation (on the right) is crucial to avoid this bottleneck

Coverage and opportunities

- Appropriate for high-level programming models.
  - OpenMP, OmpSs, Cilk,…
  - Mixing scheduling/synchronization and application code is limited.
  - Runtime system can be used as the *dynamic component*.
- Not suitable for:
  - Scheduling dependent on user code (user-guided scheduling).
  - Computation based on *random* values (e.g., Monte Carlo algorithms).

- Runtime system development:
  - Scheduling policies.
  - Overall efficiency optimizations.
  - For future machines before the actual hardware is available.
- Runtime software/hardware co-design.
  - Hardware support for runtime system.

Conclusions

- We propose a novel trace-driven simulation methodology for multithreaded applications.

- The methodology is based on distinguishing:
  - Application intrinsic behavior (user code).
  - Parallelism-management operations (*parops*).

- It allows to properly simulate different architecture configurations:
  - With different numbers of cores.
  - Using a single trace per application.

- It provides a framework not only for architecture exploration but also for runtime system development.