Efficient Multiprogramming for Multicores with SCAF

Timothy Creech, Aparna Kotha, Rajeev Barua

University of Maryland, College Park, MD

DEPARTMENT OF
ELECTRICAL &
COMPUTER ENGINEERING

This work was supported by a NASA Office of the Chief Technologist's Space Technology Research Fellowship.

Overview

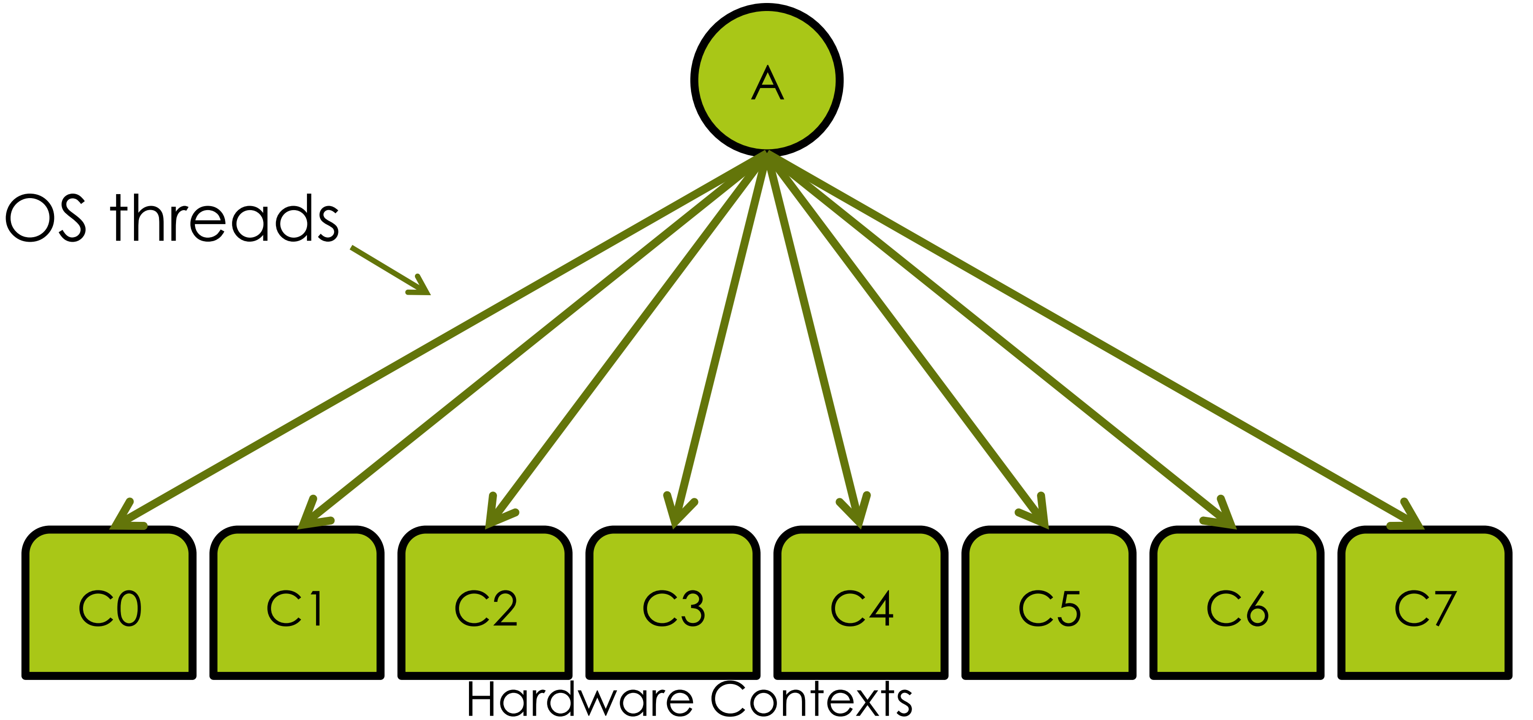
- Introduction
  - Current Systems
  - Objective
- Techniques
  - Dynamic Allocation Adjustment
  - Sum-speedup Allocation Policy
  - Serial “Experiments”
- Evaluation
  - Summary
  - Example Scenario

Current Systems

- Increasingly parallel processors
- Increasingly parallel software
- Multiprogramming parallel processes is difficult
  - Current operating systems schedule **threads**, not processes
  - The problem of managing parallelism is largely left to the user
- **Multiprogramming**: running multiple processes simultaneously

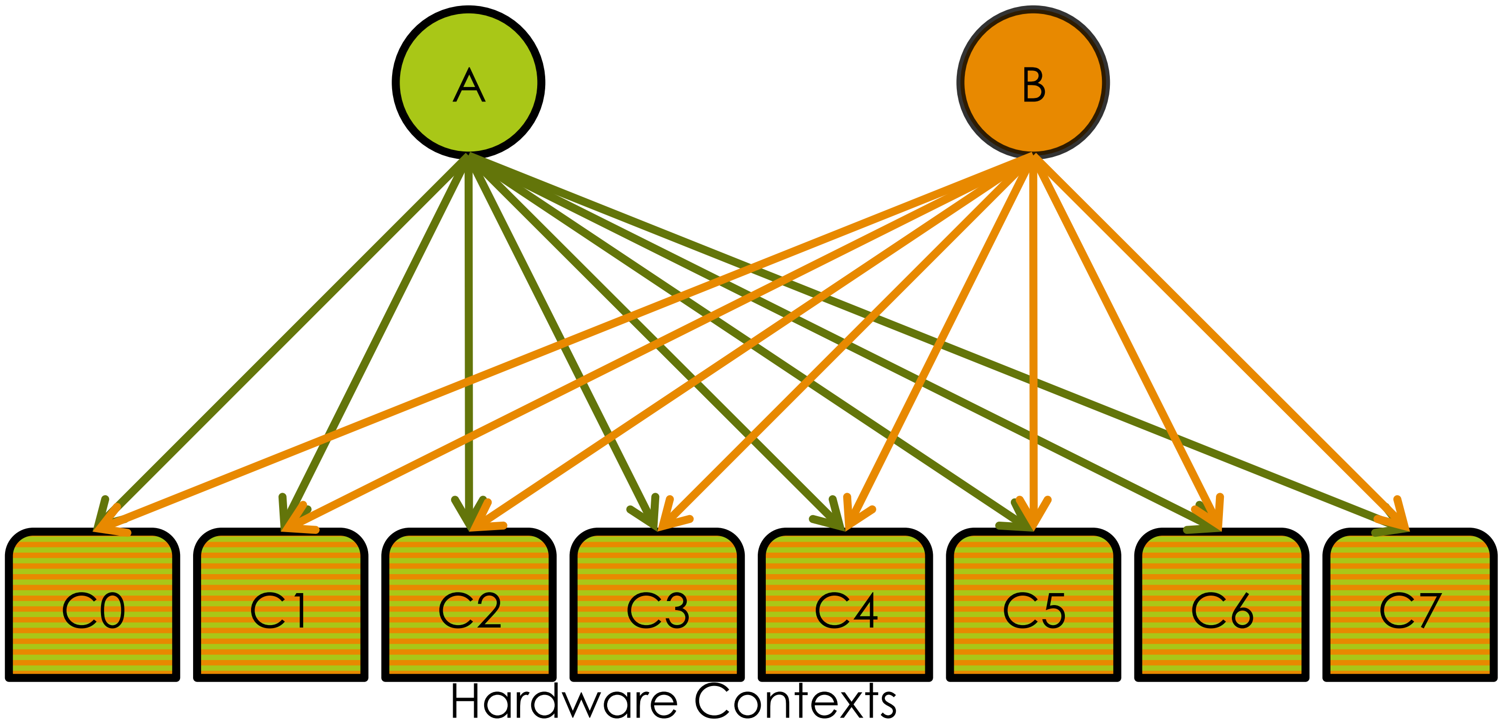
Current Systems

- Consider a single parallel application "A" running alone
- The OS schedules 8 threads to 8 cores – excellent!



Current Systems

- What if a second parallel process exists?
- The OS has to share 8 cores among *16* busy threads
  - The machine is now *oversubscribed*

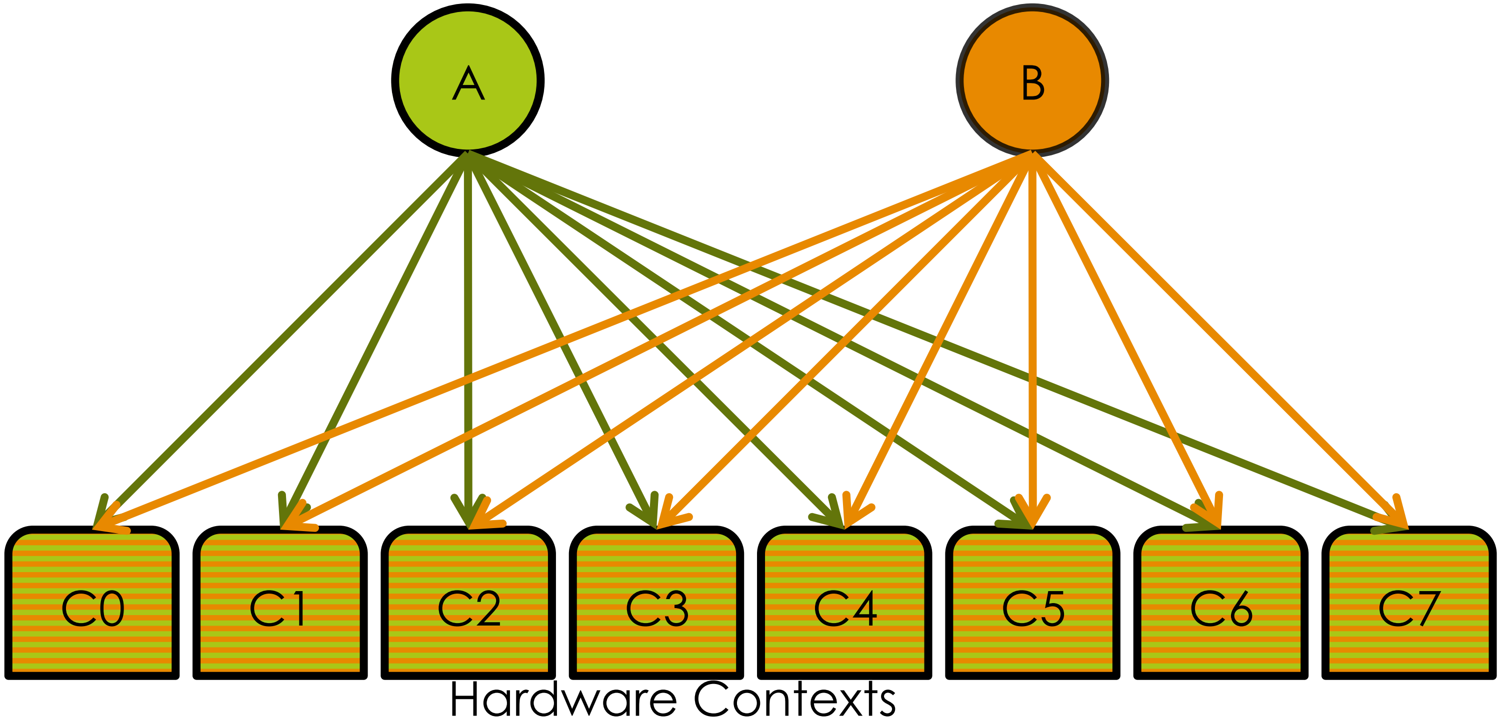


Current Systems

- What’s the problem with oversubscription?
  - Unnecessarily reduces granularity
  - Increased overhead by context switching
  - Hardware contention
  - Certain synchronization constructs assume no oversubscription

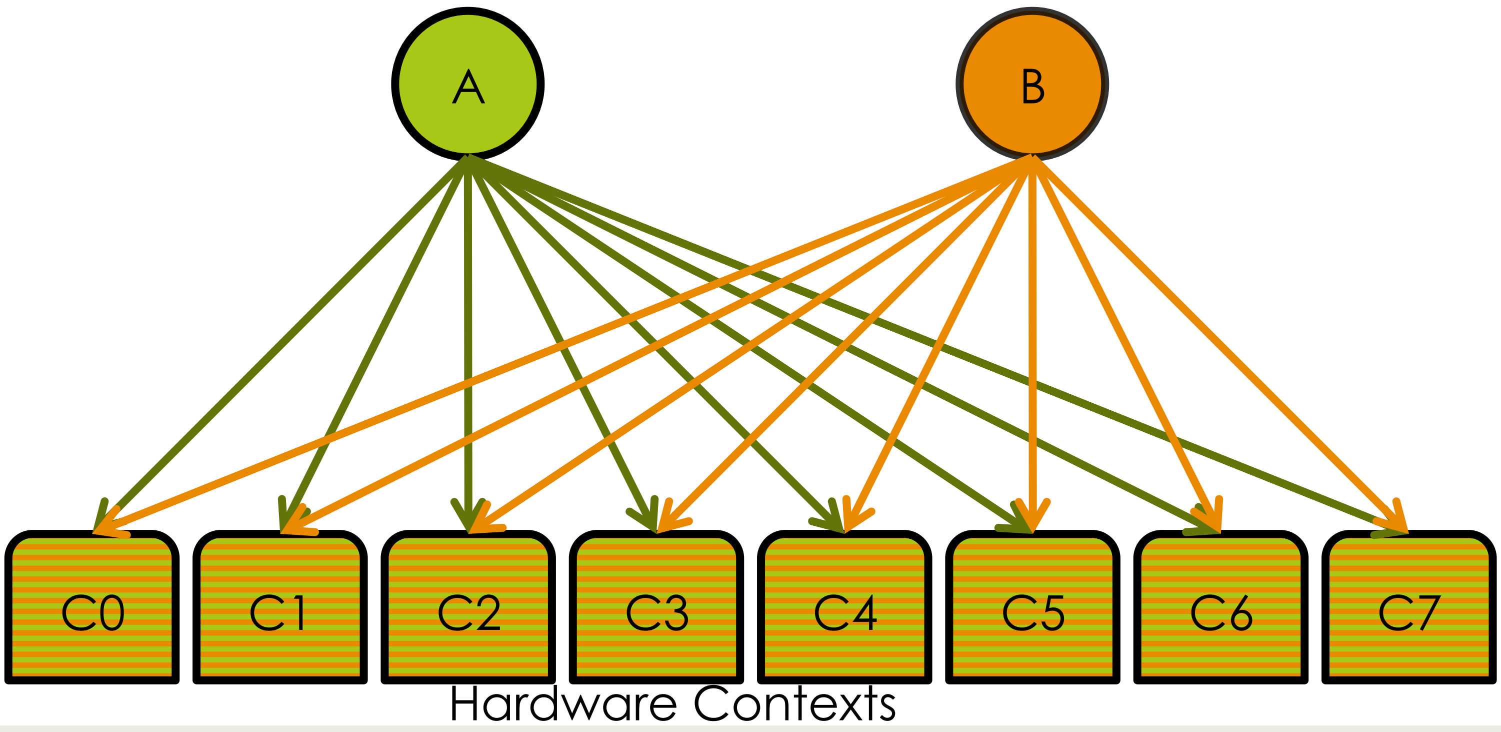
Current Systems

- What can we do?



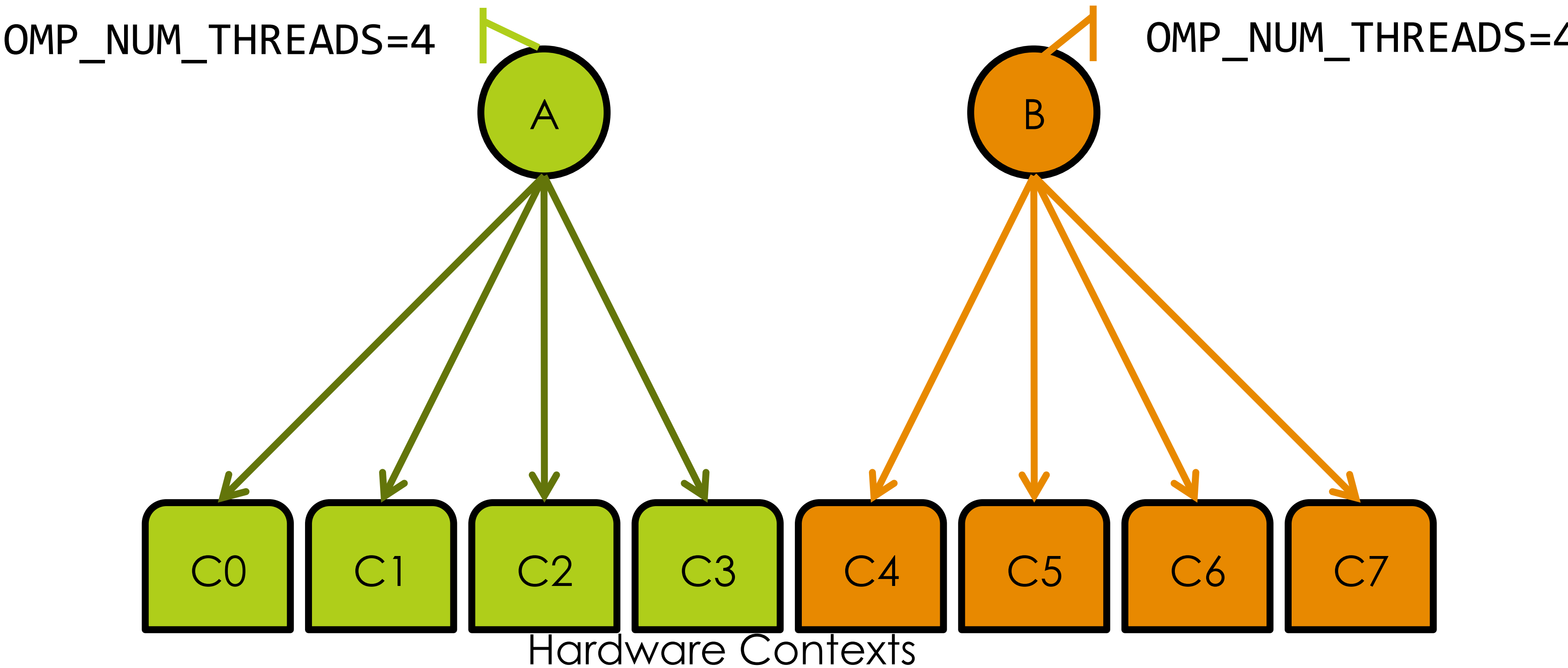
Current Systems

- What can we do?
- Make the thread scheduler's job easier
  - Avoid oversubscription with **space sharing**



Current Systems

- What can we do?
- Make the thread scheduler's job easier
  - Avoid oversubscription with **space sharing**



Current Systems

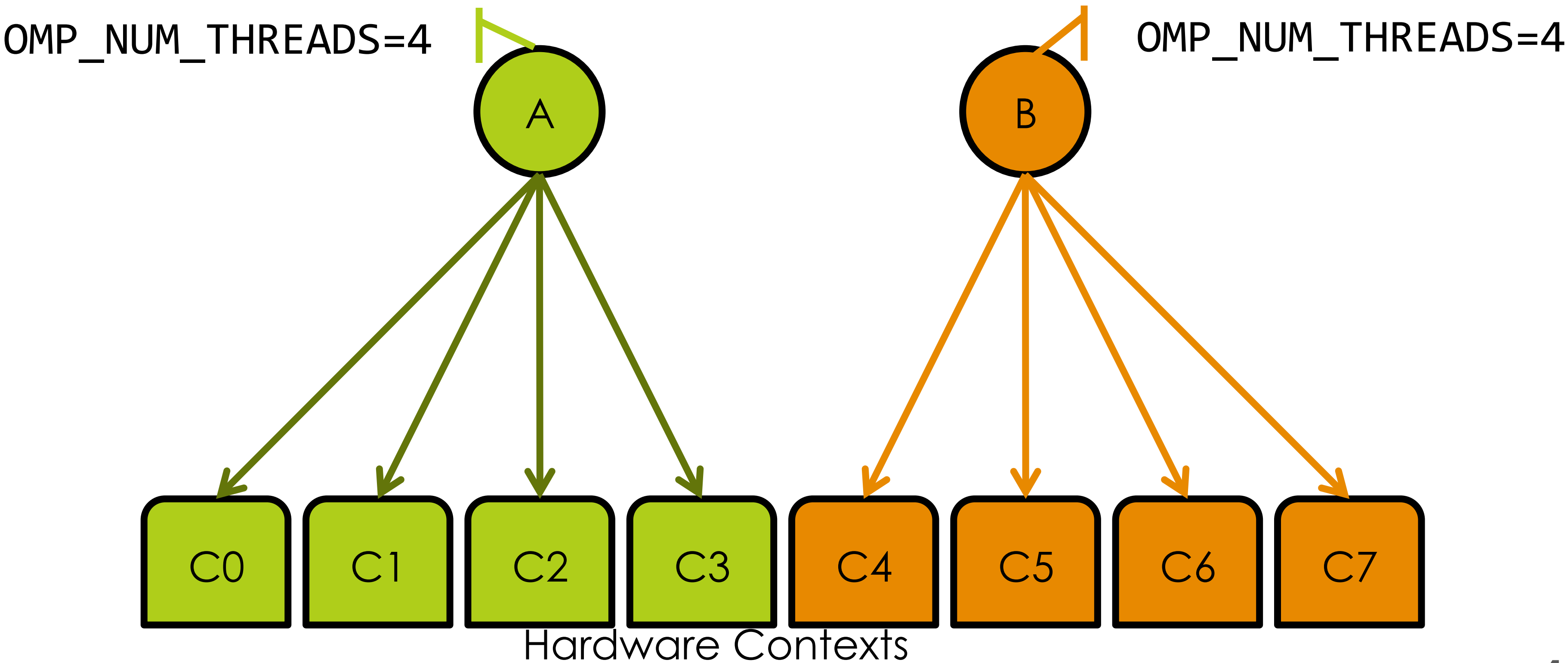
- Example: NAS SP+LU on 8-core Xeon E5410
  - Unmodified system: oversubscription
  - Equipartitioned space sharing: Each process gets half of the machine

System	Threads	SP Speedup	LU Speedup	Σ Speedup
Unmodified	8+8=16	1.18	1.78	**2.96**
Equi-partitioning	4+4=8	1.33	2.84	**4.17**

- System 40% more efficient with equipartitioning

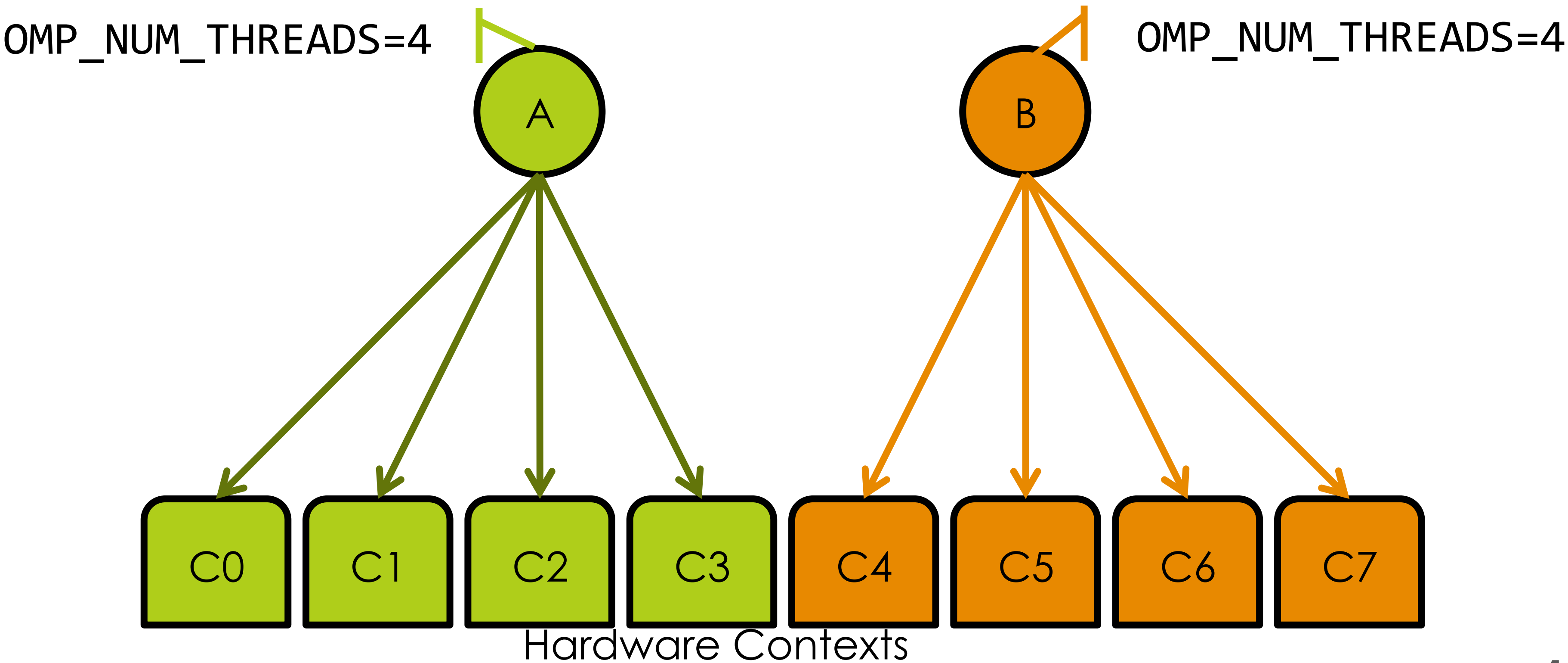
Current Systems

- This is a lot to ask of users
  - Must be aware of the parallel runtime
  - Must manually coordinate processes
  - **More difficult than serial multiprogramming**



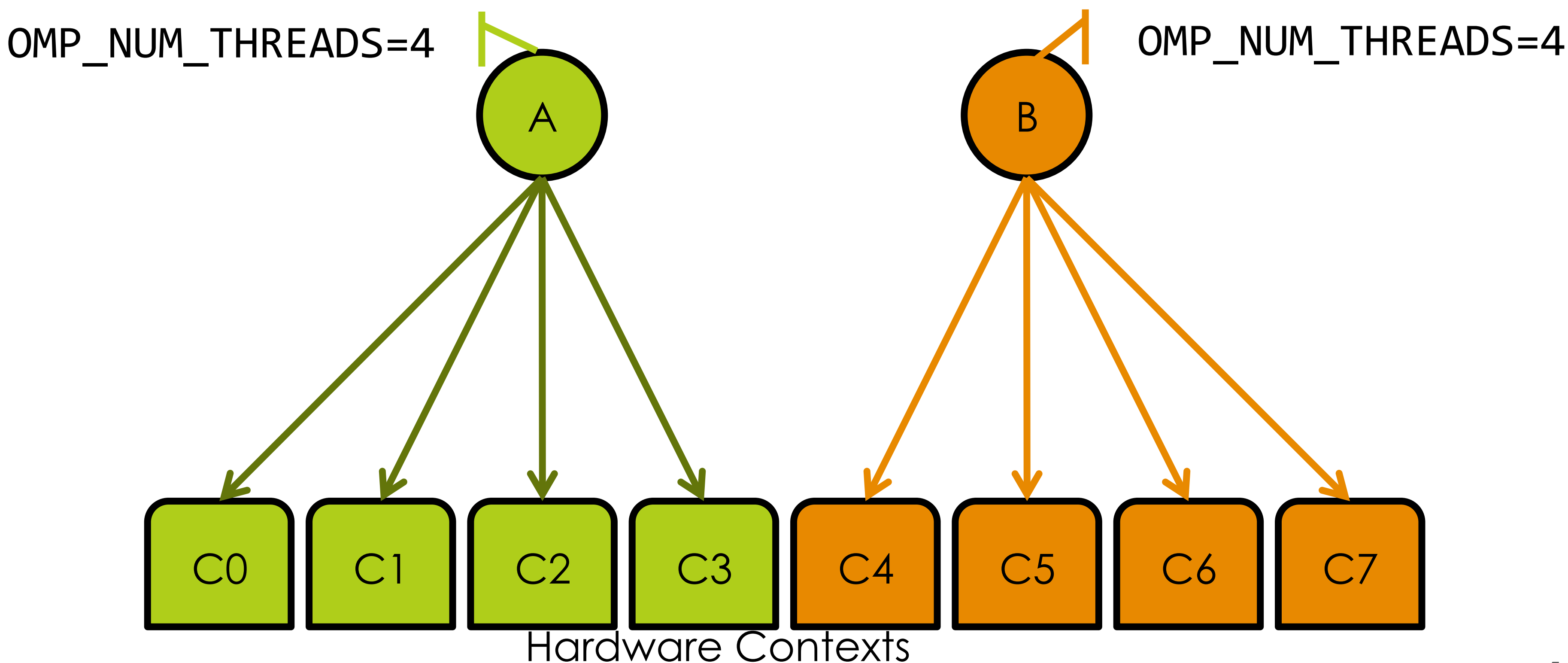
Current Systems

- Equipartitioning is rare in practice
  - Exists in literature, no popular solution
  - Difficult to do manually as a user



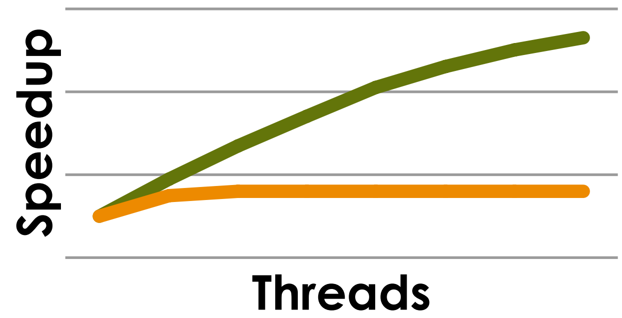
Current Systems

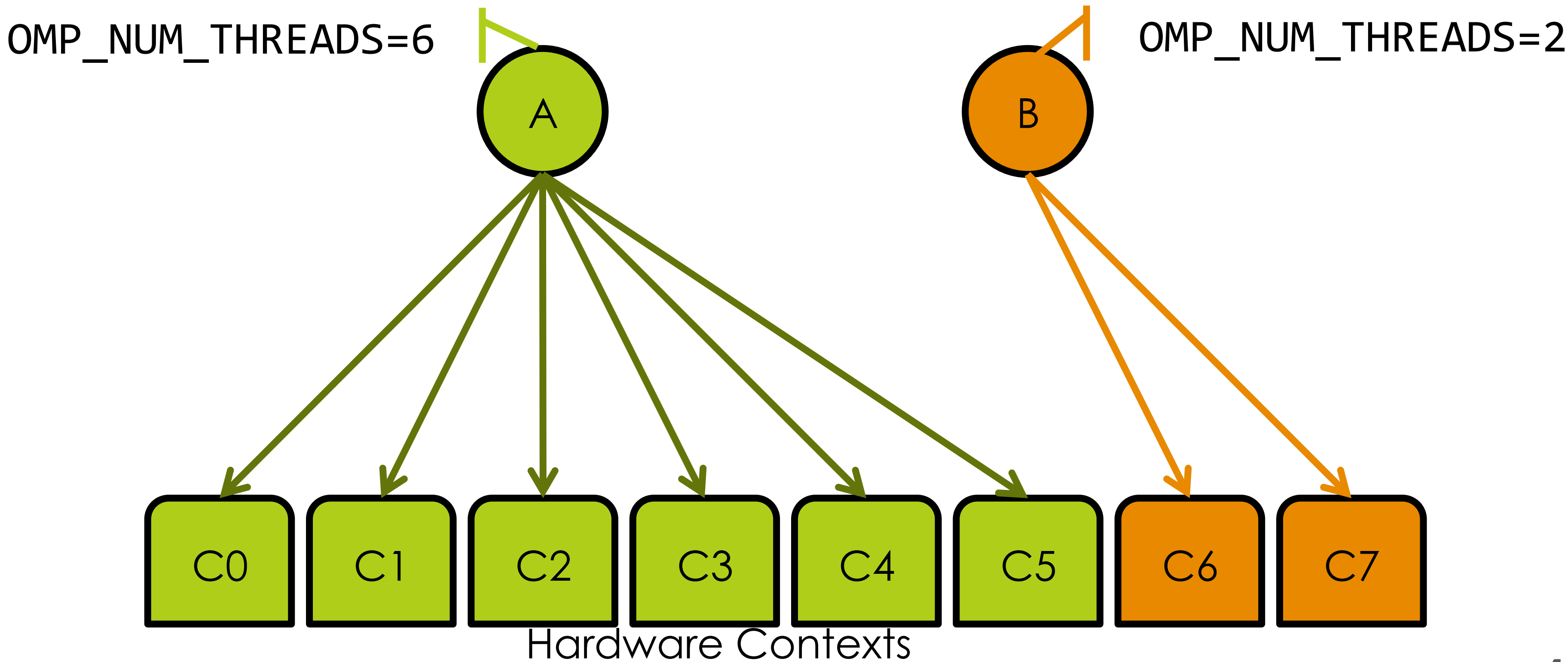
- Now, say "A" gets good speedups
- But "B" has limited parallelism, speeds up significantly less



Current Systems

- Allow A to use more threads:
  - make better use of hardware contexts
  - Sum of speedups is improved





Current Systems

- Example: NAS SP+LU on 8-core Xeon E5410
  - Unmodified system: oversubscription
  - Equipartitioned space sharing: Each process gets half of the machine

System	Threads	SP Speedup	LU Speedup	Σ Speedup
Unmodified	8+8=16	1.18	1.78	**2.96**
Equi-partitioning	4+4=8	1.33	2.84	**4.17**
“Smart” partitioning	2+6=8	1.26	4.81	**6.07**

- System another **45%** more efficient with intelligent partitioning

Objective

"SCAF"

(**Sc**heduling and **A**llocation with **F**eedback)

1. Improve **system efficiency**
   - Attempt to maximize the *sum of speedups*
   - No oversubscription: dynamic space sharing
   - Allocate based on run-time feedback
   - Achieve higher efficiency than equipartitioning

Objective

"SCAF"

(Scheduling and Allocation with Feedback)

2. **Convenience** for users
   - Automatic partitioning of contexts
   - No offline profiling
   - No porting, modification, or recompilation of binaries
   - As simple as serial multiprogramming

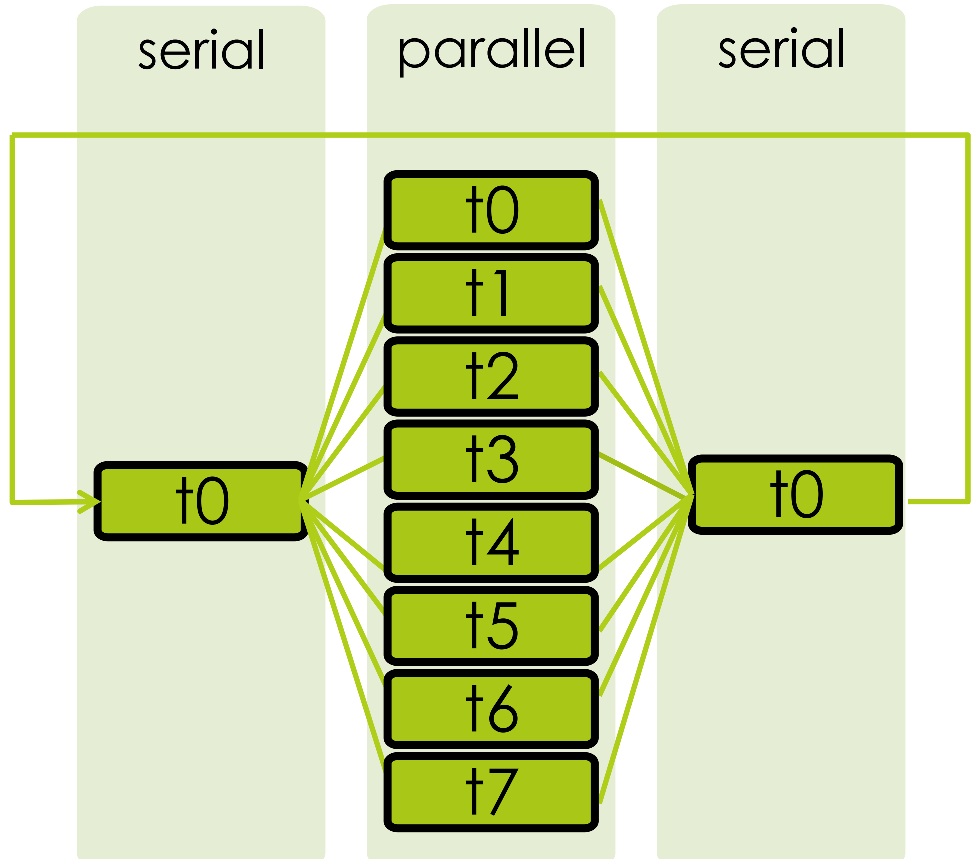
Techniques

- Allocation Adjustment
- Sum-speedup Allocation Policy
- Serial “Experiments”

Dynamic Allocation Adjustment

- OpenMP code generation example:

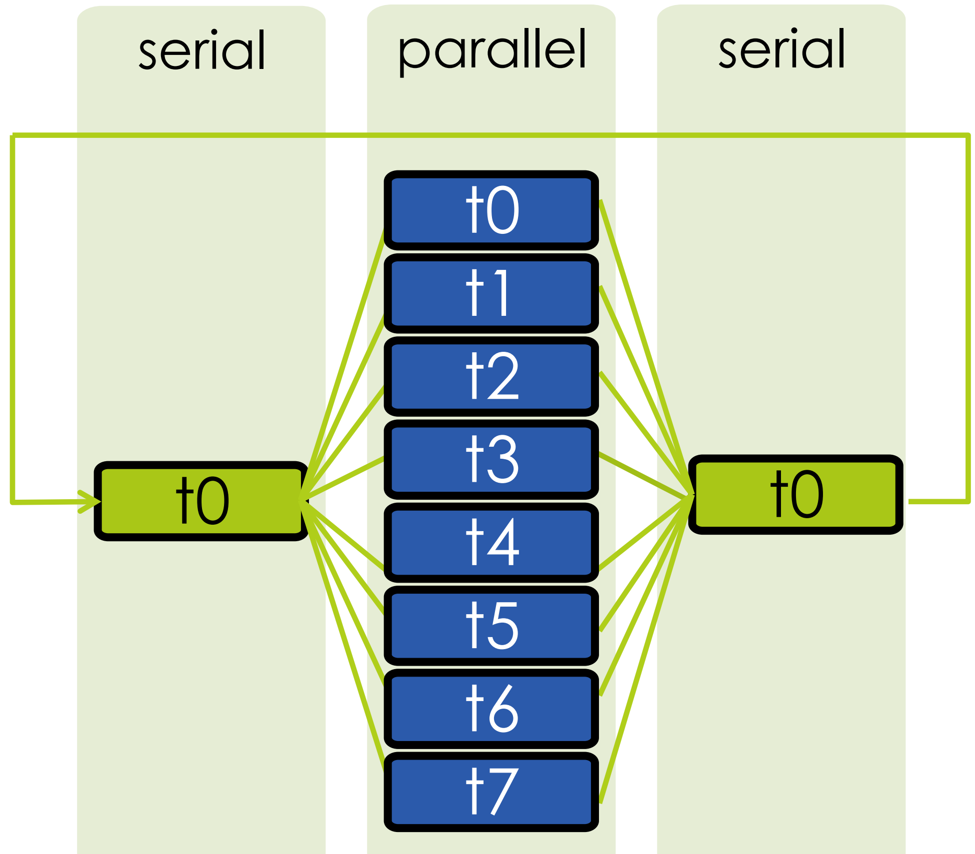
```
unsigned time, I;
for(time=0; time<TMAX; time++){

#pragma omp parallel for
        for(i=0; i<IMAX; i++)
                process(i,data);

        aggregate(data);
}
```



Dynamic Allocation Adjustment

- OpenMP code generation example:
  - Parallel runtime (GOMP) spawns threads according to hardware

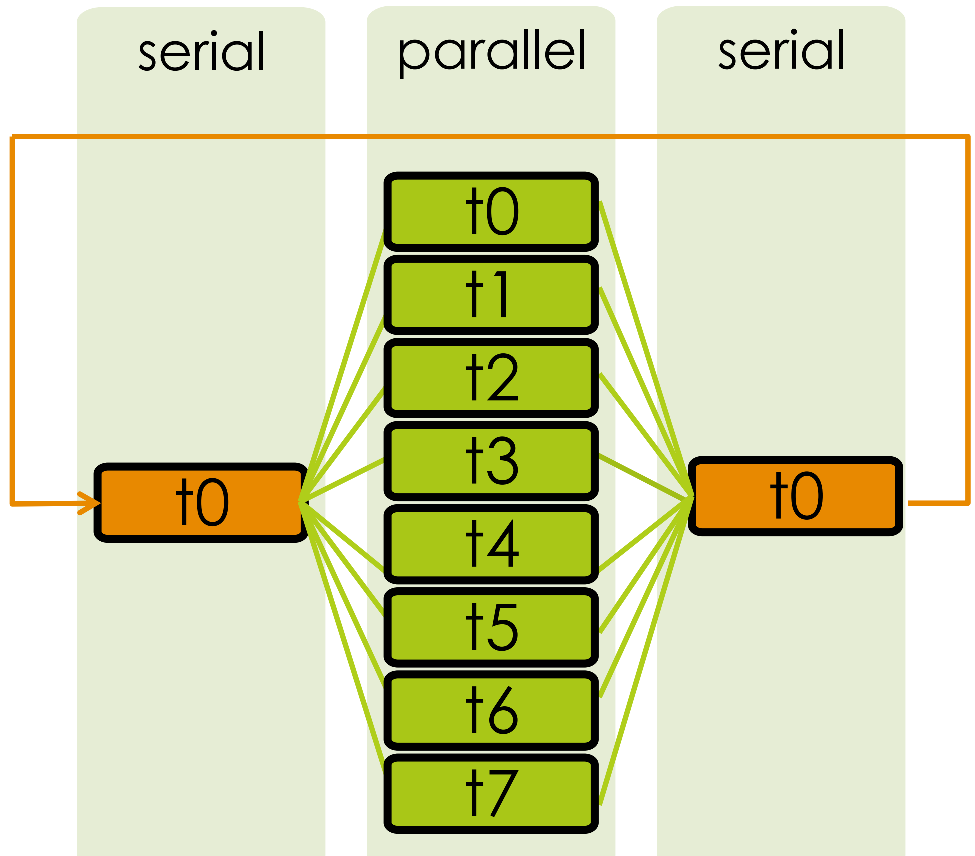
```
unsigned time, I;
for(time=0; time<TMAX; time++){

#pragma omp parallel for
        for(i=0; i<IMAX; i++)
                process(i,data);

        aggregate(data);
}
```



Dynamic Allocation Adjustment

- OpenMP code generation example:
  - Program also spends some time in serial sections between parallel sections

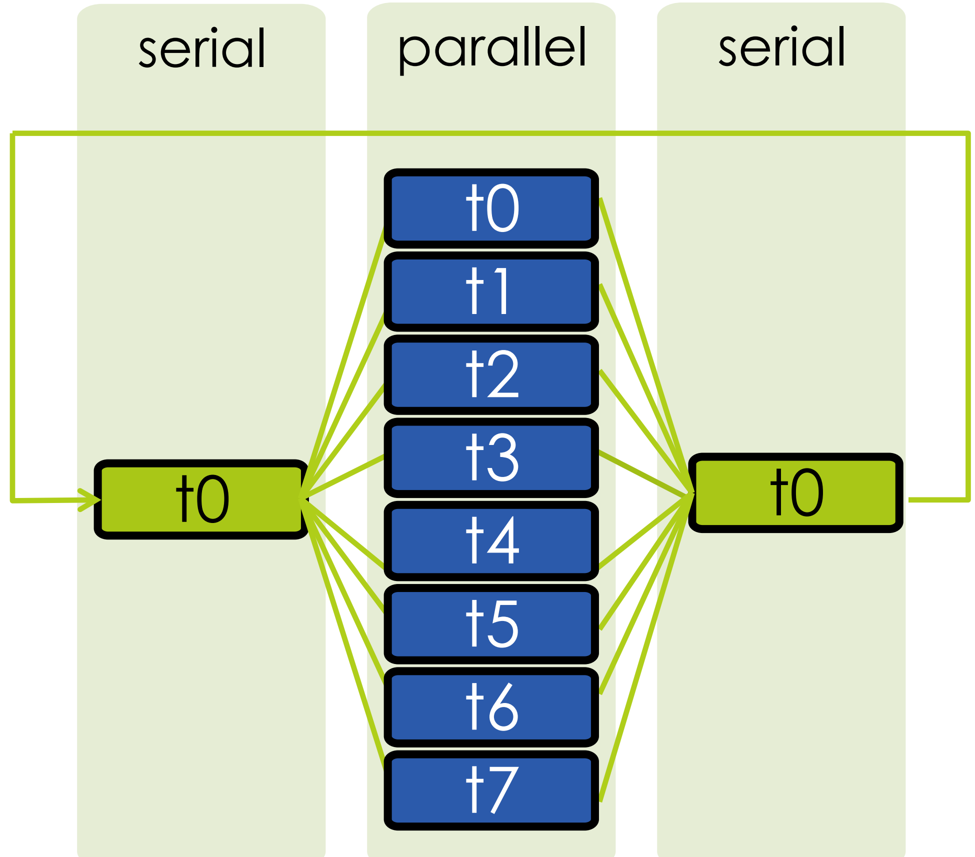
```
unsigned time, I;
for(time=0; time<TMAX; time++){

#pragma omp parallel for
        for(i=0; i<IMAX; i++)
                process(i,data);

        aggregate(data);
}
```



Dynamic Allocation Adjustment

- OpenMP code generation example:
  - Spawn/join parallelism
  - Note: serial/parallel phases always alternate

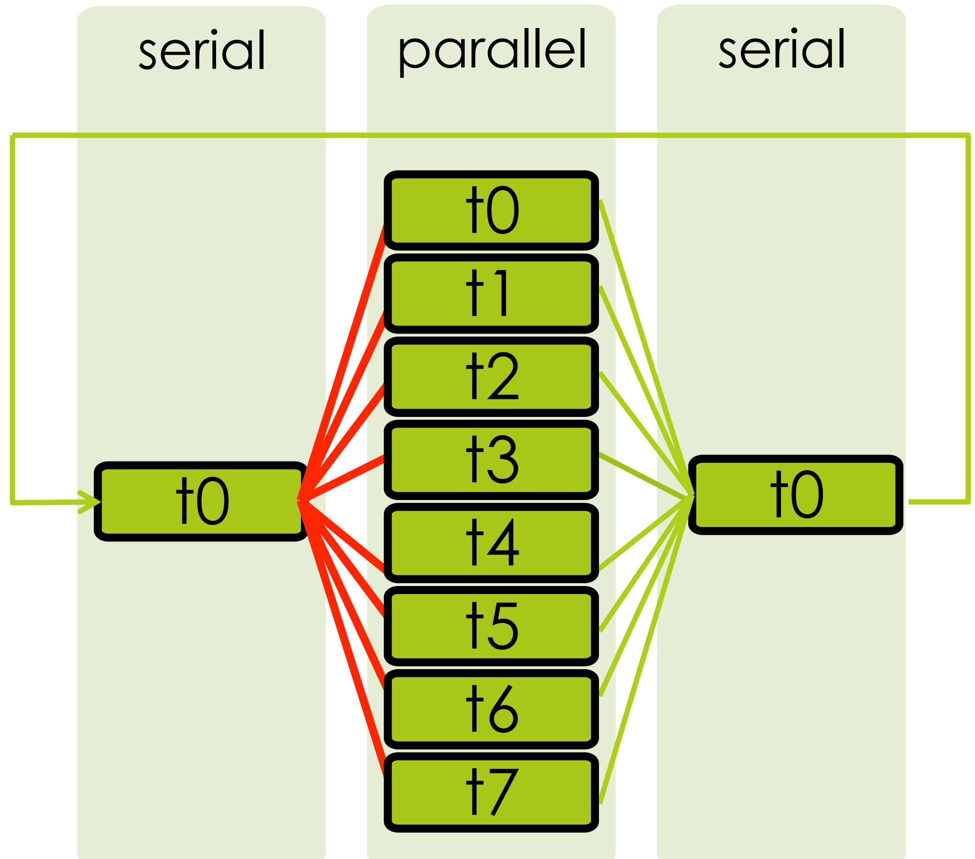
```
unsigned time, I;
for(time=0; time<TMAX; time++){

#pragma omp parallel for
        for(i=0; i<IMAX; i++)
                process(i,data);

        aggregate(data);
}
```



Dynamic Allocation Adjustment

- The parallel loop is **malleable**
  - Can run on *any number* of threads

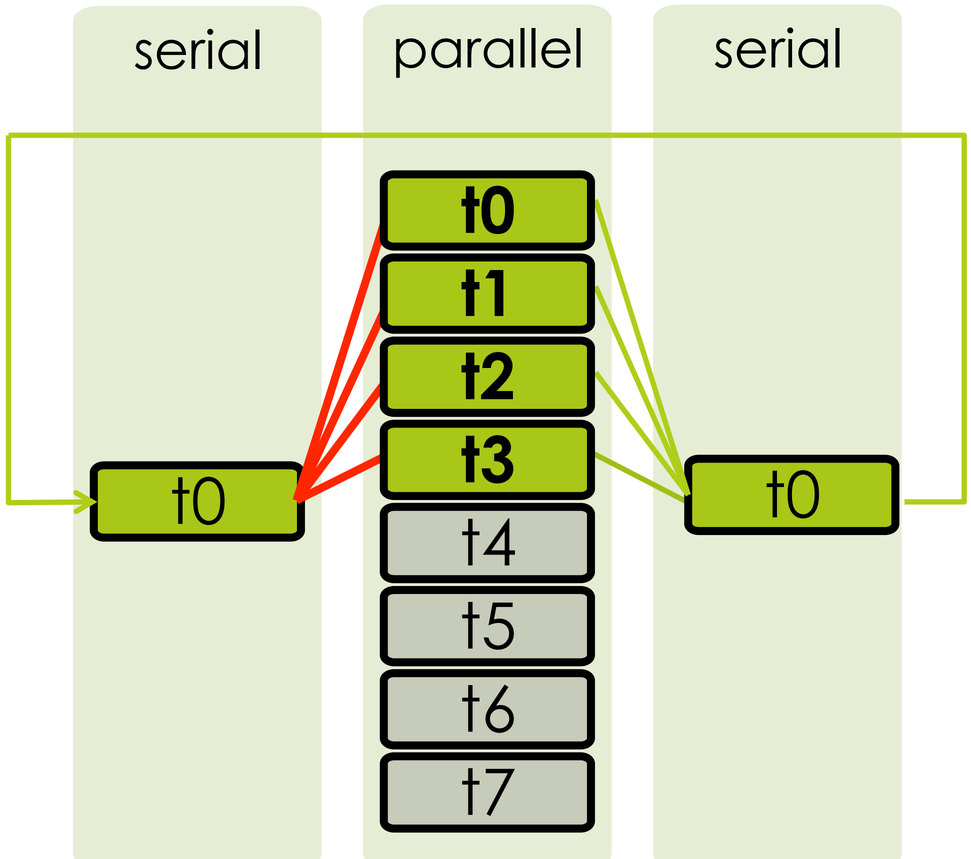
```
unsigned time, I;
for(time=0; time<TMAX; time++){

#pragma omp parallel for
        for(i=0; i<IMAX; i++)
                process(i,data);

        aggregate(data);
}
```



Dynamic Allocation Adjustment

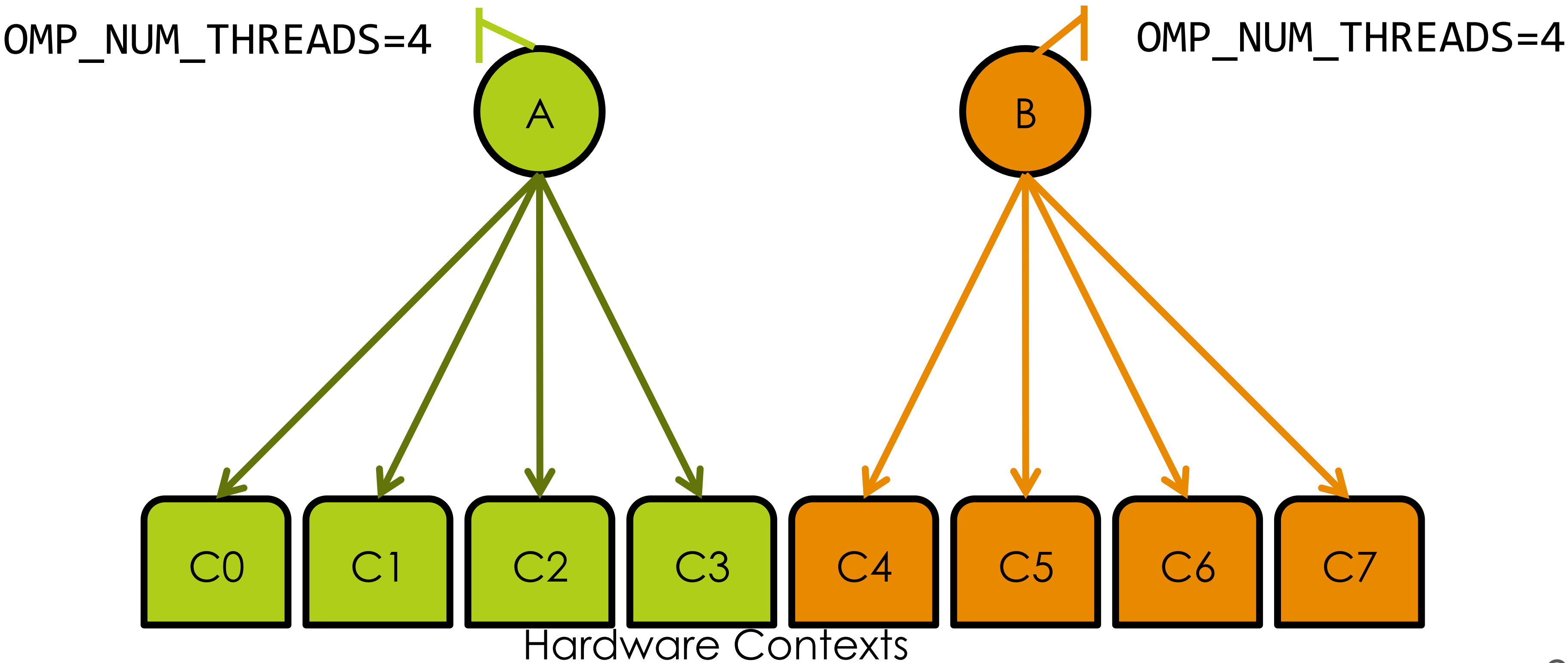
- Modify OpenMP runtime library
  - Control parallelism of **malleable** loops

```
unsigned time, I;
for(time=0; time<TMAX; time++){

#pragma omp parallel for
        for(i=0; i<IMAX; i++)
                process(i,data);

        aggregate(data);
}
```



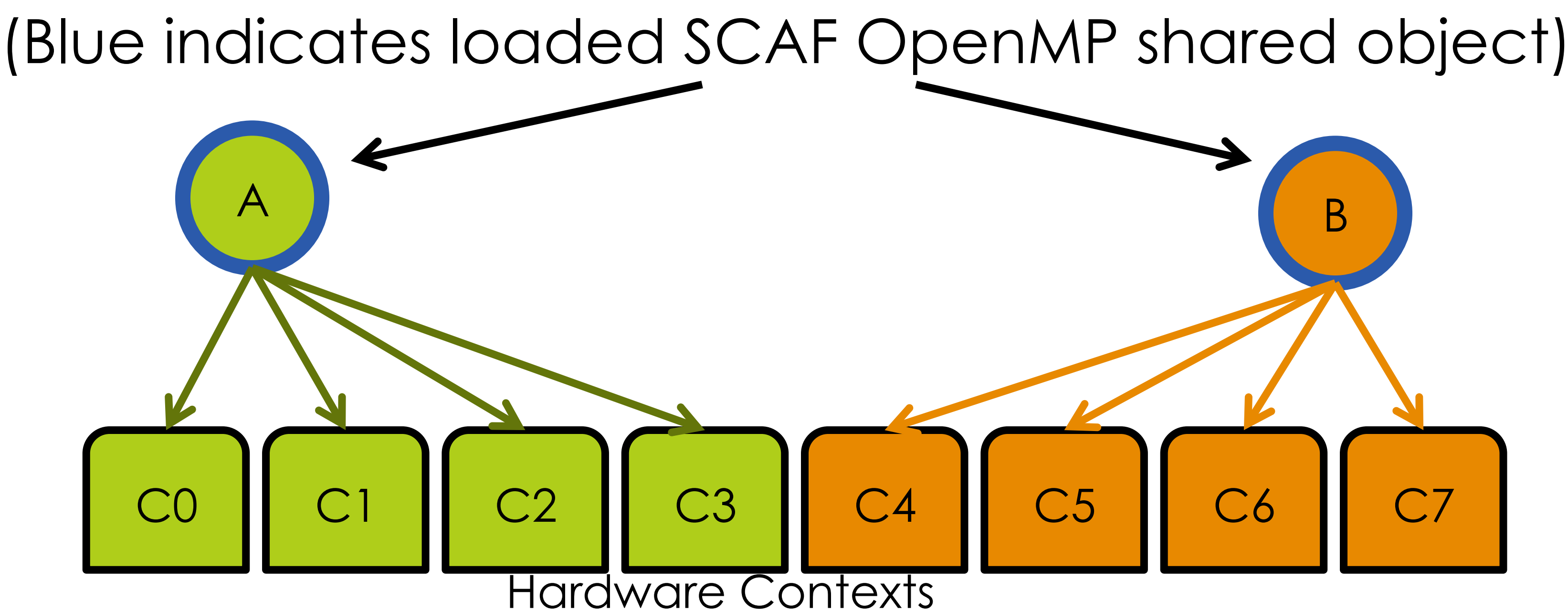
Dynamic Allocation Adjustment

- Instead of users manually (and statically) space sharing…



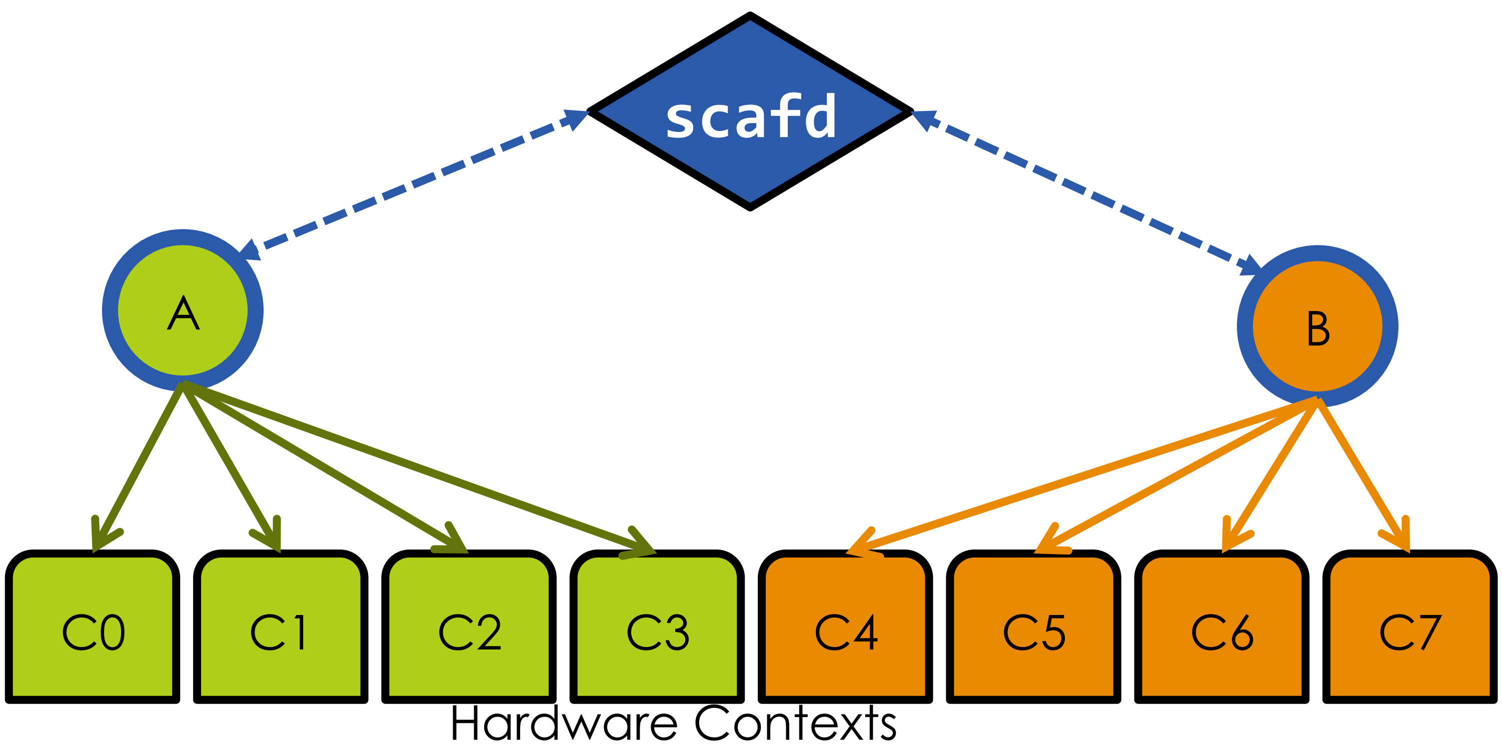
Dynamic Allocation Adjustment

- Instead of users manually and statically space sharing
  - Processes use a SCAF port of the OpenMP runtime



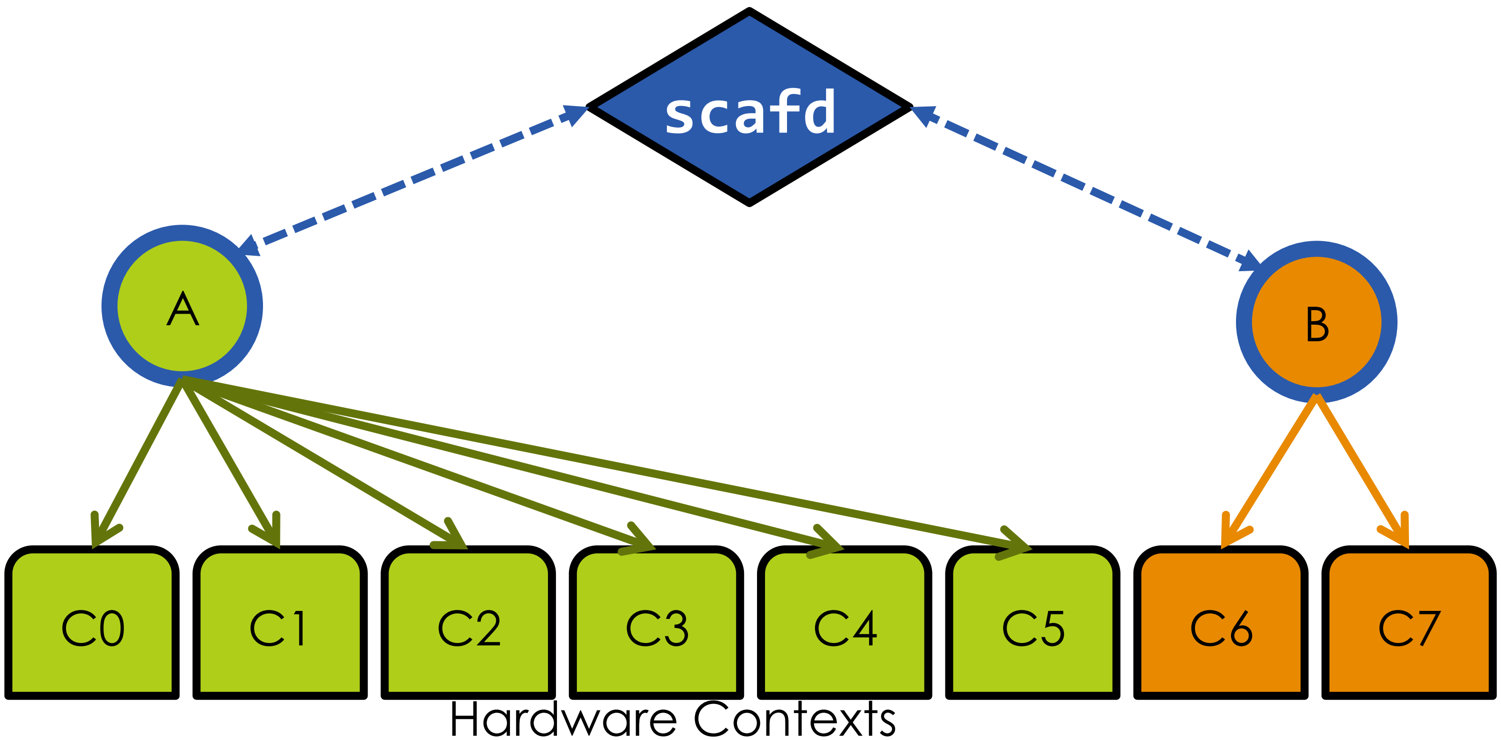
Dynamic Allocation Adjustment

- The runtime consults a SCAF daemon
  - `scafd` is system-wide and centralized
  - Userspace only – no kernel modifications



Dynamic Allocation Adjustment

- `scafd` can make allocation decisions dynamically
  - Allocation decisions respected at each process's next parallel spawn



Dynamic Allocation Adjustment

- Note: processes A and B are **unaware** of SCAF
- Only the compiler-dependent parallel runtime communicates with the SCAF daemon
- Enable SCAF runtimes with OS's dynamic linker/loader

Sum-speedup Allocation Policy

- Program content and completion time are unknown
  - The daemon (`scafd`) can only reason about instantaneous performance, e.g., IPC, Flop/s
  - Most general work metric: speedup over serial execution
  - Make partitioning decisions which attempt to maximize the **sum of speedups**
- Each parallel process reports **efficiency** metric to `scafd`

$$efficiency = {}^{speedup}\!\big/_{threads}$$

Sum-speedup Allocation Policy

- `scafd` uses this single efficiency metric (E) to fit each process to a simple sub-linear speedup function S(p):

$$S(p) \approx 1 + C \cdot \log(p), \quad \text{where } C \leftarrow \frac{E \cdot p' - 1}{\log(p')}$$

- Log-speedup chosen for single coefficient and diminishing returns

Sum-speedup Allocation Policy

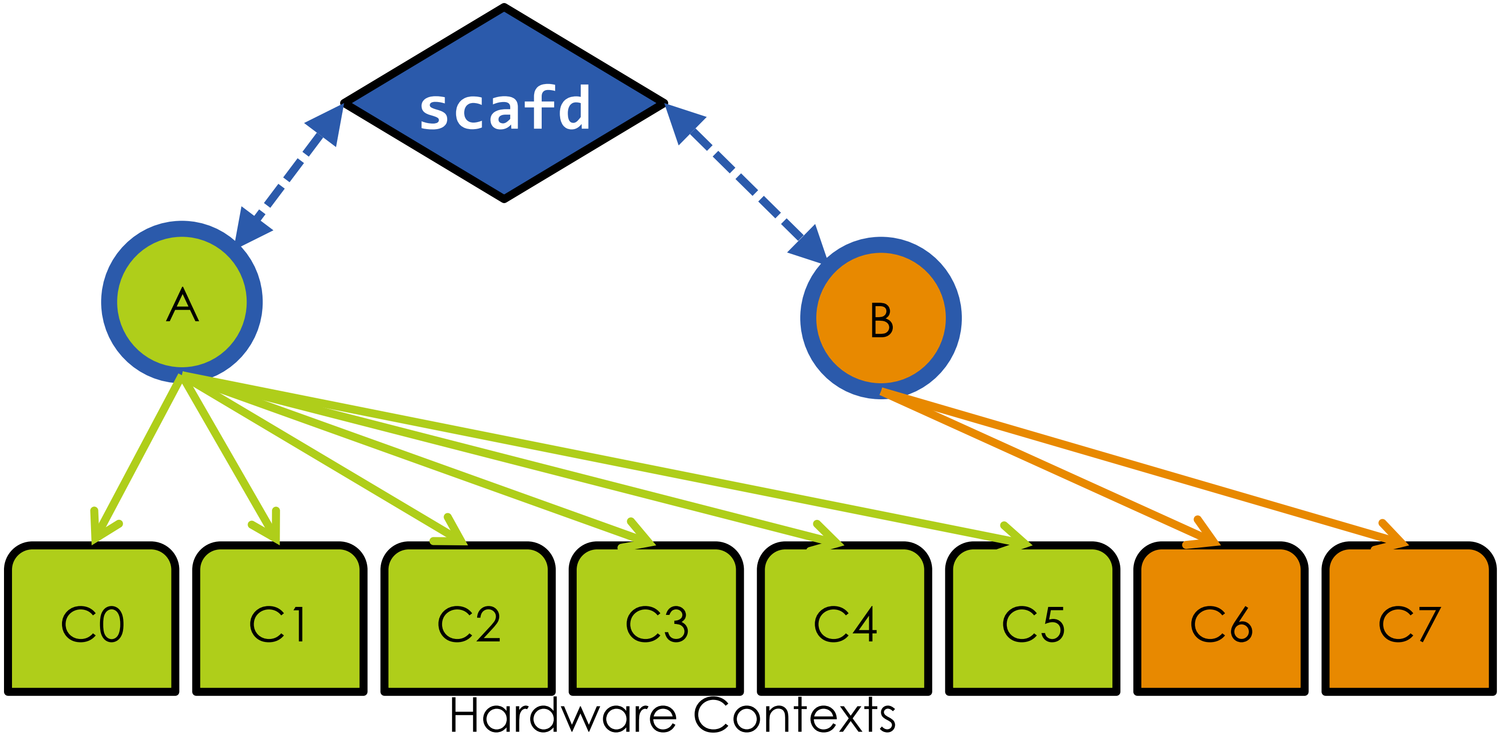
- This has an intuitive closed solution:

$$p_i \leftarrow \left.\frac{N \cdot C_i}{\sum_j C_j}\right\} \text{where} \quad \begin{array}{l} N \text{ is the number of processes} \\ C_k \text{ is the fitting coefficient of process k} \\ p_k \text{ is the allocation of processes k} \end{array}$$

- In other words, each process receives an allocation proportional to its fitting coefficient

Sum-speedup Allocation Policy

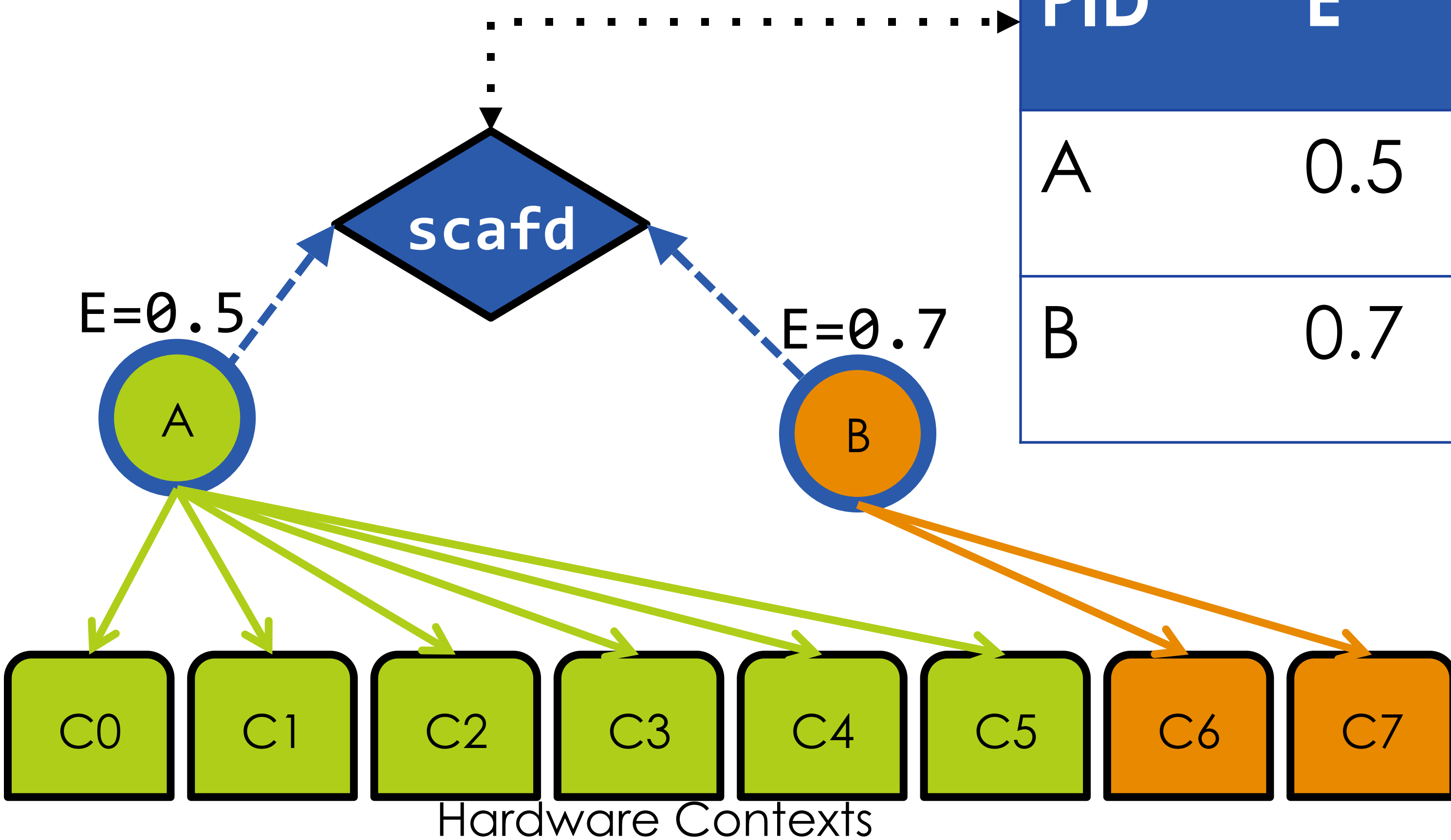
- scafd uses this policy to advise processes on allocations



Sum-speedup Allocation Policy

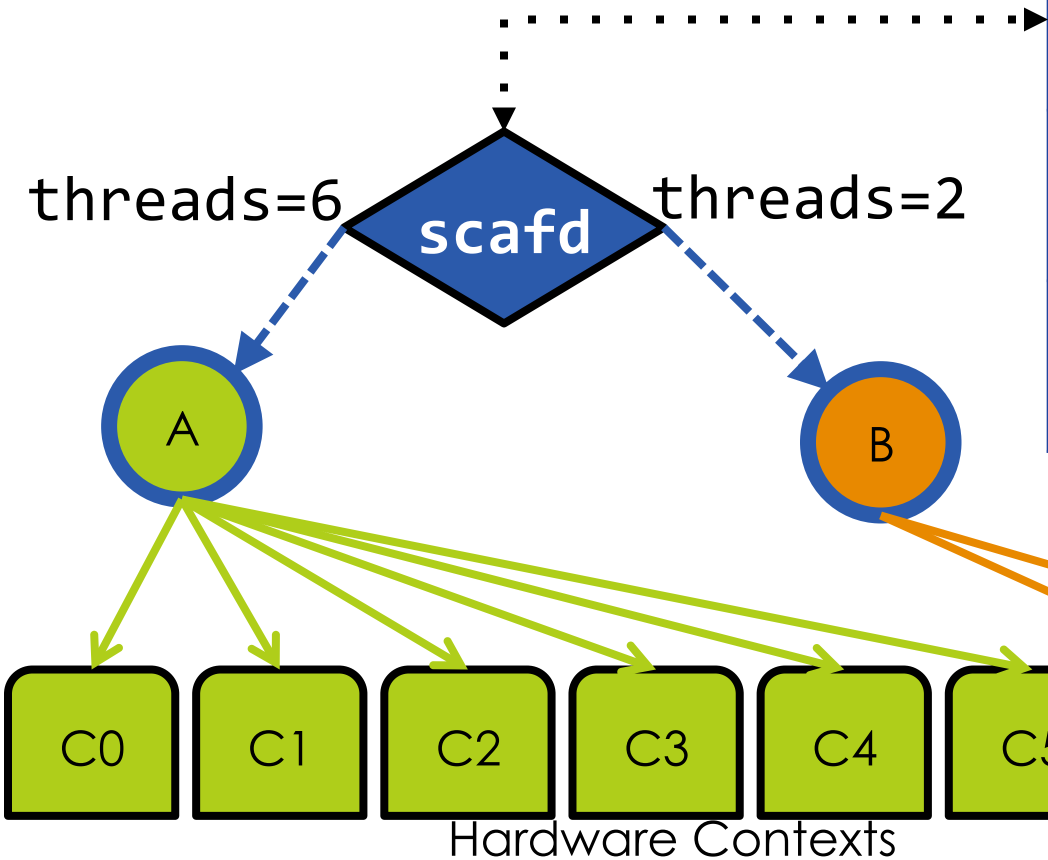
- Processes report efficiencies after parallel sections

PID	E	Threads
A	0.5	**6**
B	0.7	**2**



Sum-speedup Allocation Policy

- scafd immediately responds with pre-computed allocations

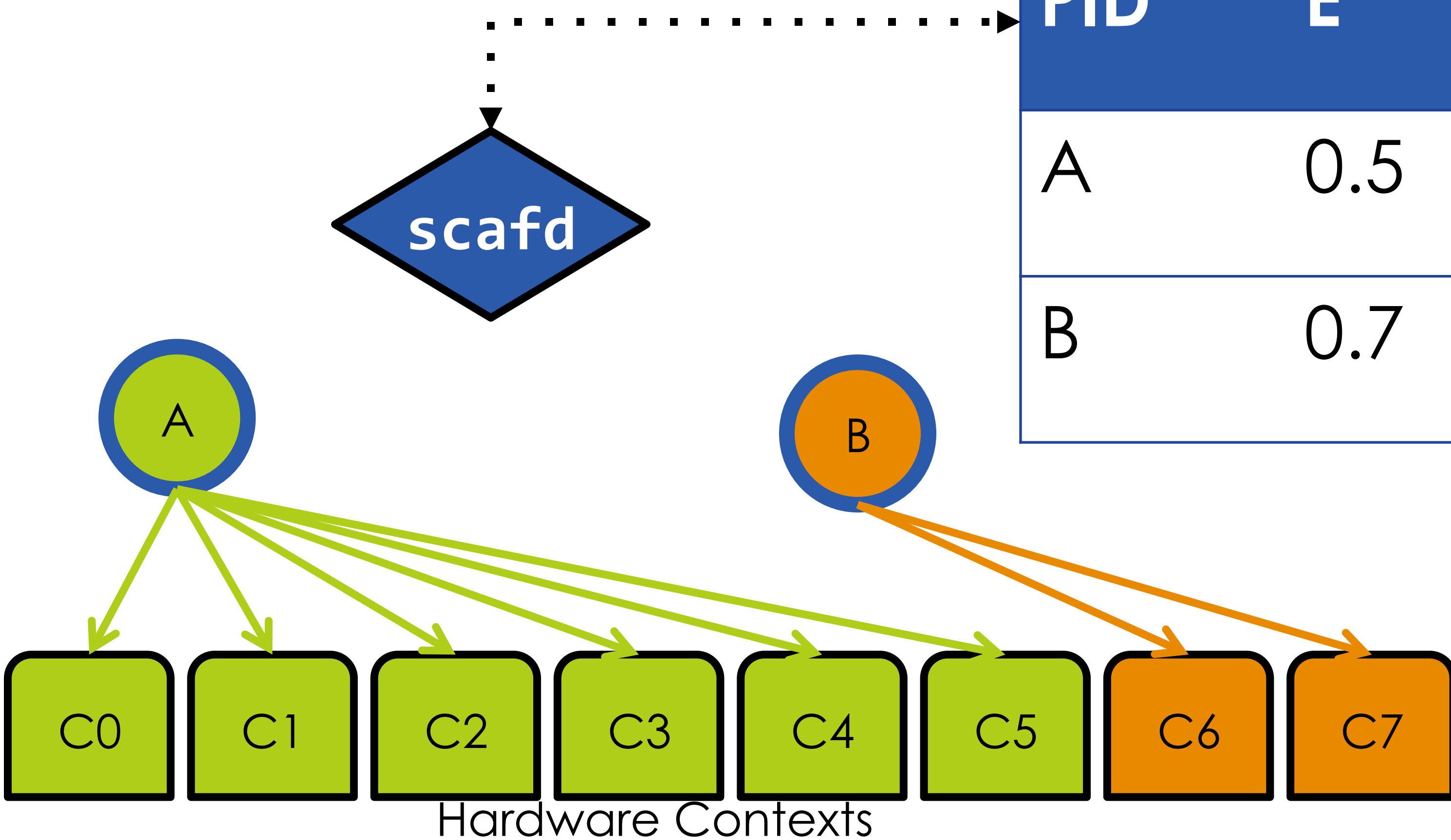


PID	E	Threads
A	0.5	**6**
B	0.7	**2**

Sum-speedup Allocation Policy

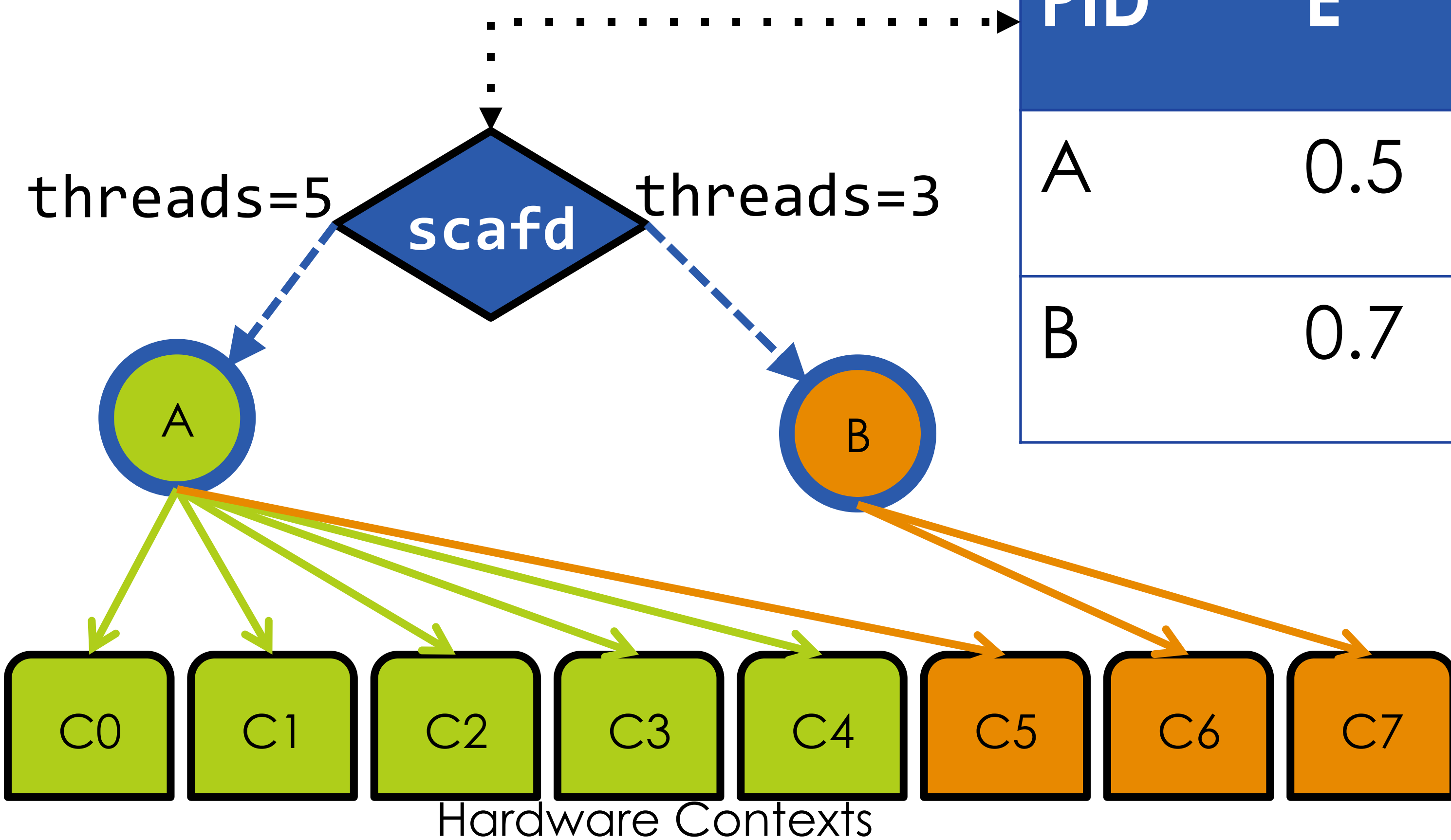
- scafd re-evaluates allocations periodically (~4Hz)

PID	E	Threads
A	0.5	**5**
B	0.7	**3**



Sum-speedup Allocation Policy

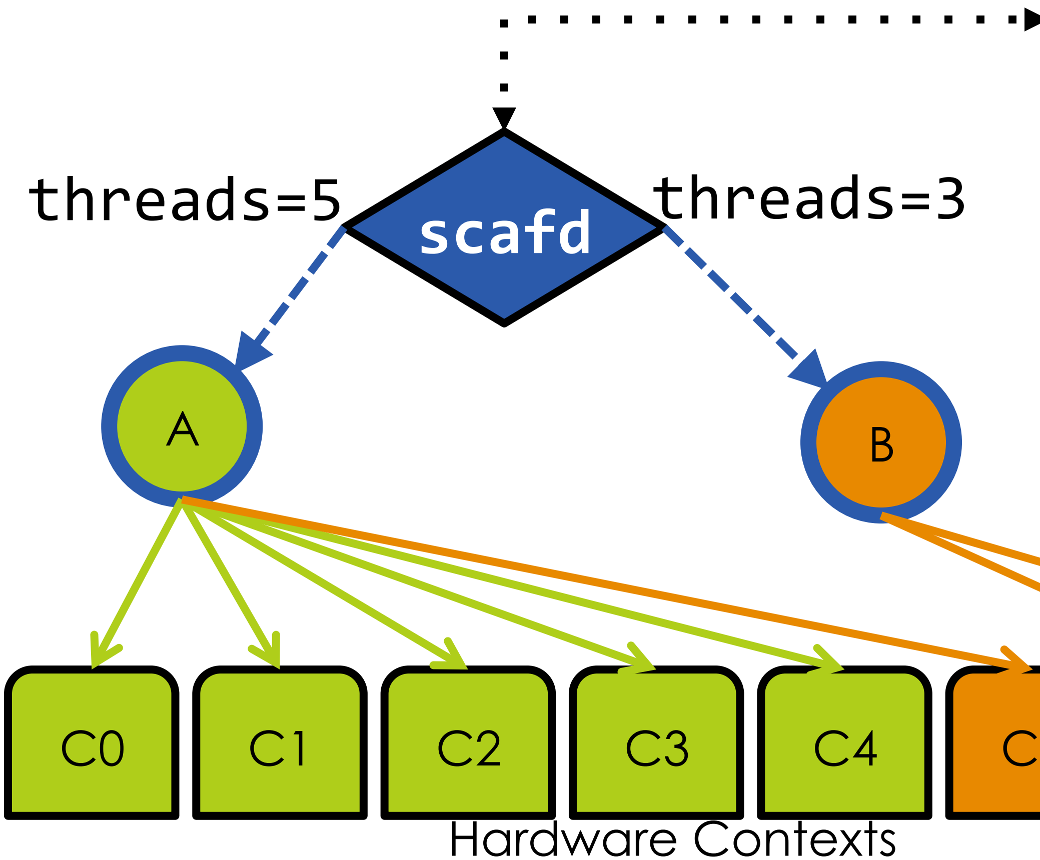
- …and communicates allocation changes



PID	E	Threads
A	0.5	**5**
B	0.7	**3**

Sum-speedup Allocation Policy

- (Lower efficiency can receive higher allocation if achieved on many threads)



PID	E	Threads
A	0.5	**5**
B	0.7	**3**

Serial Experiments

- Problem: how do processes compute their efficiency?
  - Missing information: **serial performance**
  - Don't want off-line profiling
  - Can't wait to temporarily serialize

A	A	A	A	B	B	B	B
A	A	A	A	B	B	B	B
C	C	C	C	B	B	B	B
C	C	C	C	B	B	B	B
C	C	C	C	B	B	B	B
C	C	C	C	B	B	B	B
C	C	C	C	B	B	B	B
C	C	C	C	B	B	B	B

64-core processor

Serial Experiments

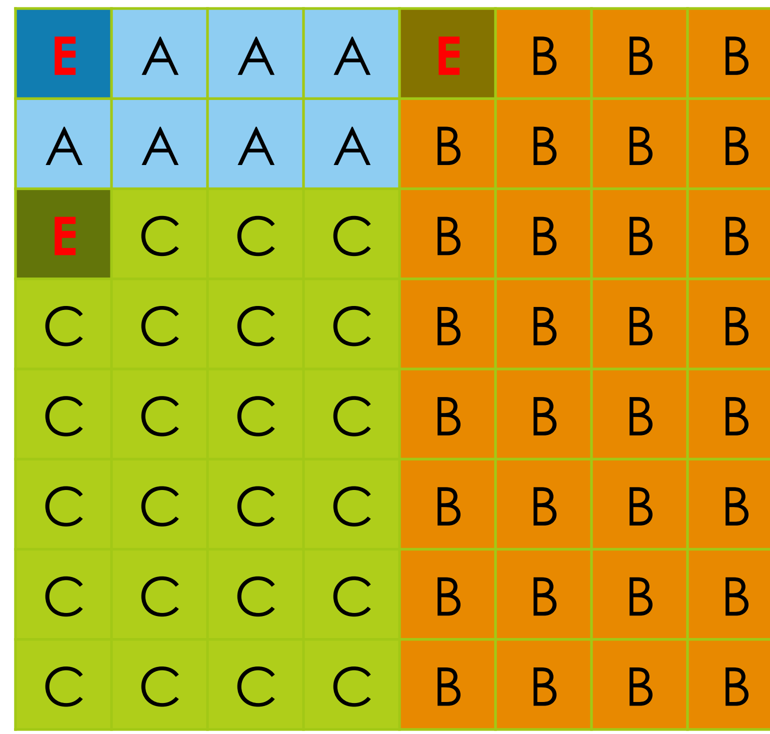
- Solution: Serial "Experiments"
  - 1 context used for experiment
  - Child process (not thread) executes the section serially
  - OS tracing facilities used to protect parent process correctness
  - Performance measured as hardware counter *rates*
  - Experiment ends early if:
    - Parallel section finishes
    - Correctness is in danger

E	A	A	A	E	B	B	B
A	A	A	A	B	B	B	B
E	C	C	C	B	B	B	B
C	C	C	C	B	B	B	B
C	C	C	C	B	B	B	B
C	C	C	C	B	B	B	B
C	C	C	C	B	B	B	B
C	C	C	C	B	B	B	B

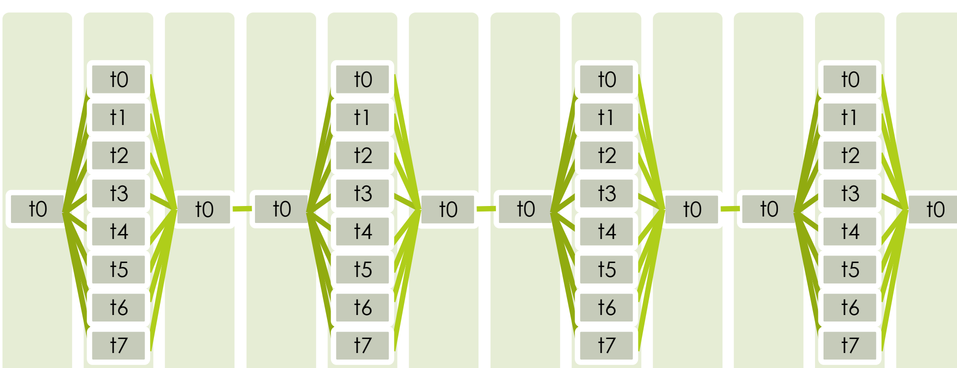
64-core processor

Serial Experiments

- Solution: Serial "Experiments"
  - Process uses 1 context to measure serial performance
  - Results used later to compute speedup/ efficiency



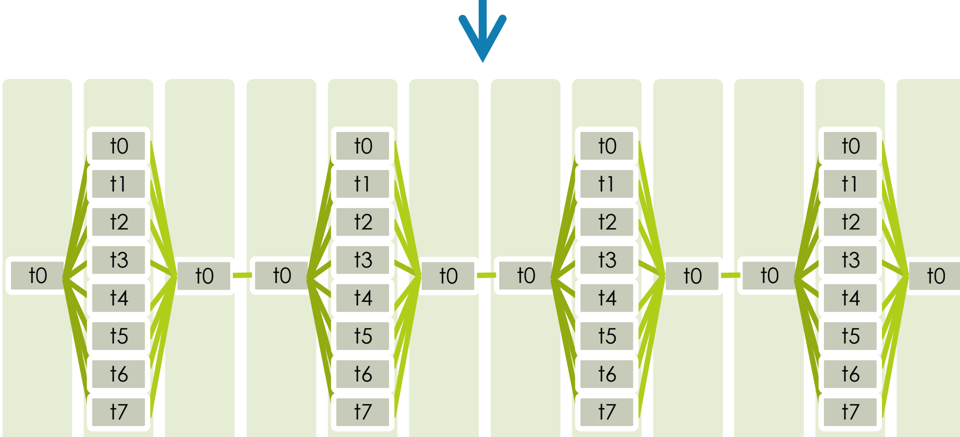
64-core processor



Unrolled execution of example loop

Serial Experiments

Example of serial experiment in process **A**:



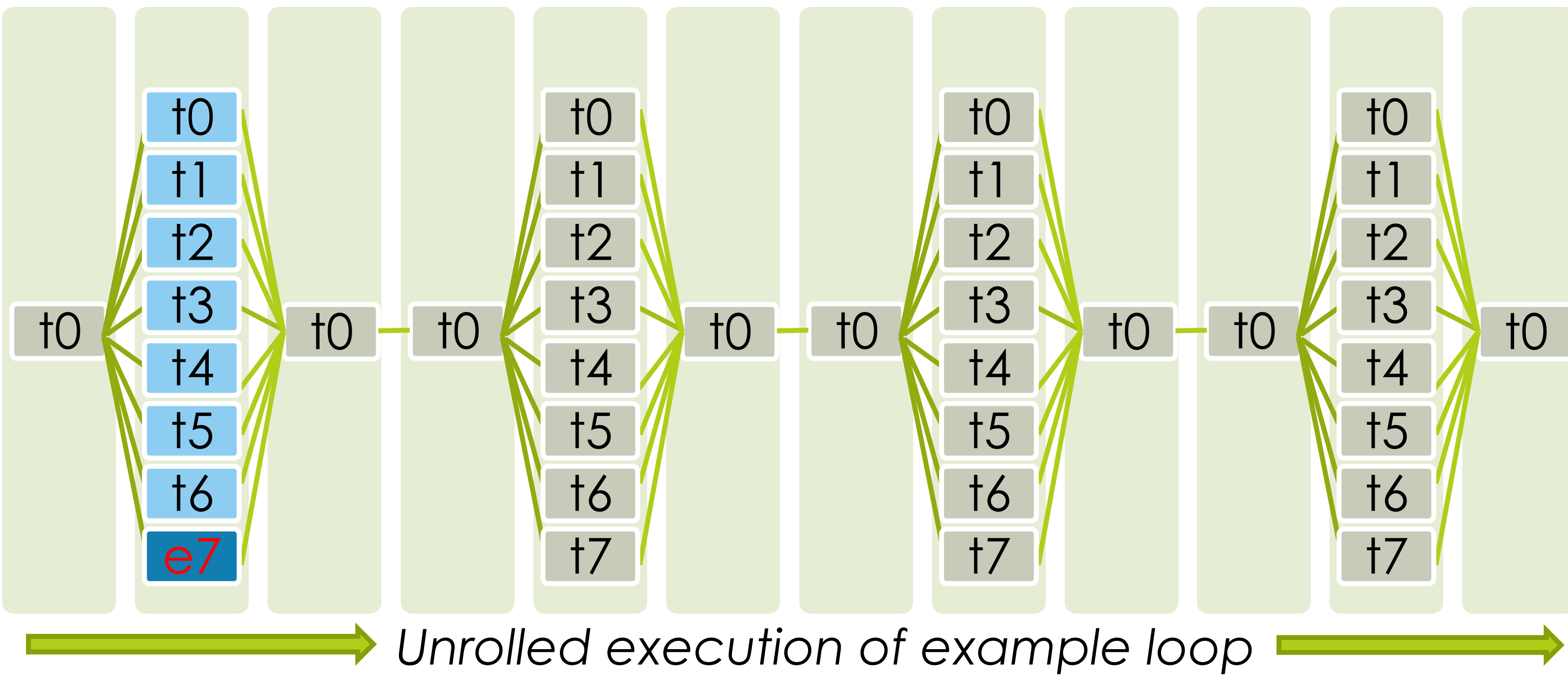
Unrolled execution of example loop

64-core processor

Serial Experiments

Example of serial experiment in process **A**:

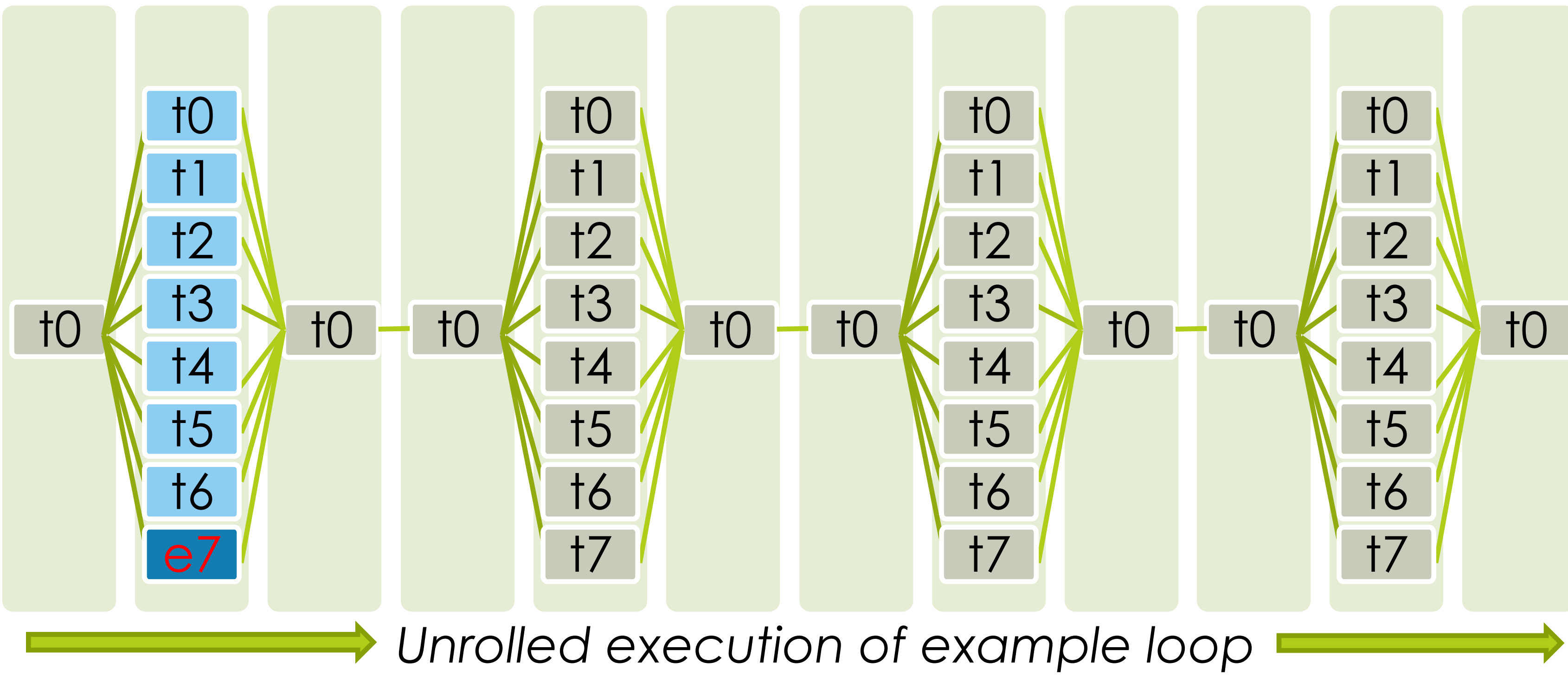
First time the parallel section is seen: use 1 core to run a serialized version



Serial Experiments

Example of serial experiment in process **A**:

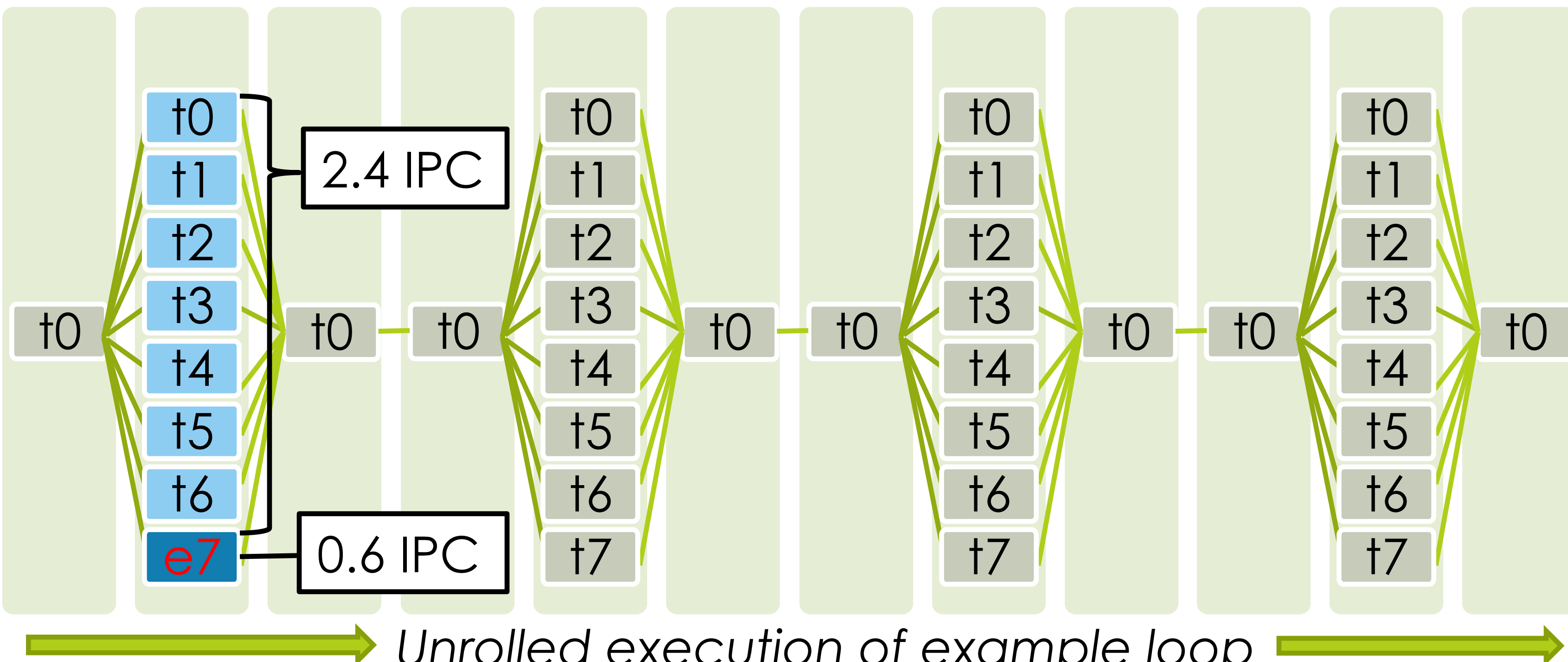
The remaining hardware contexts work on progressing through the section



Serial Experiments

Example of serial experiment in process **A**:

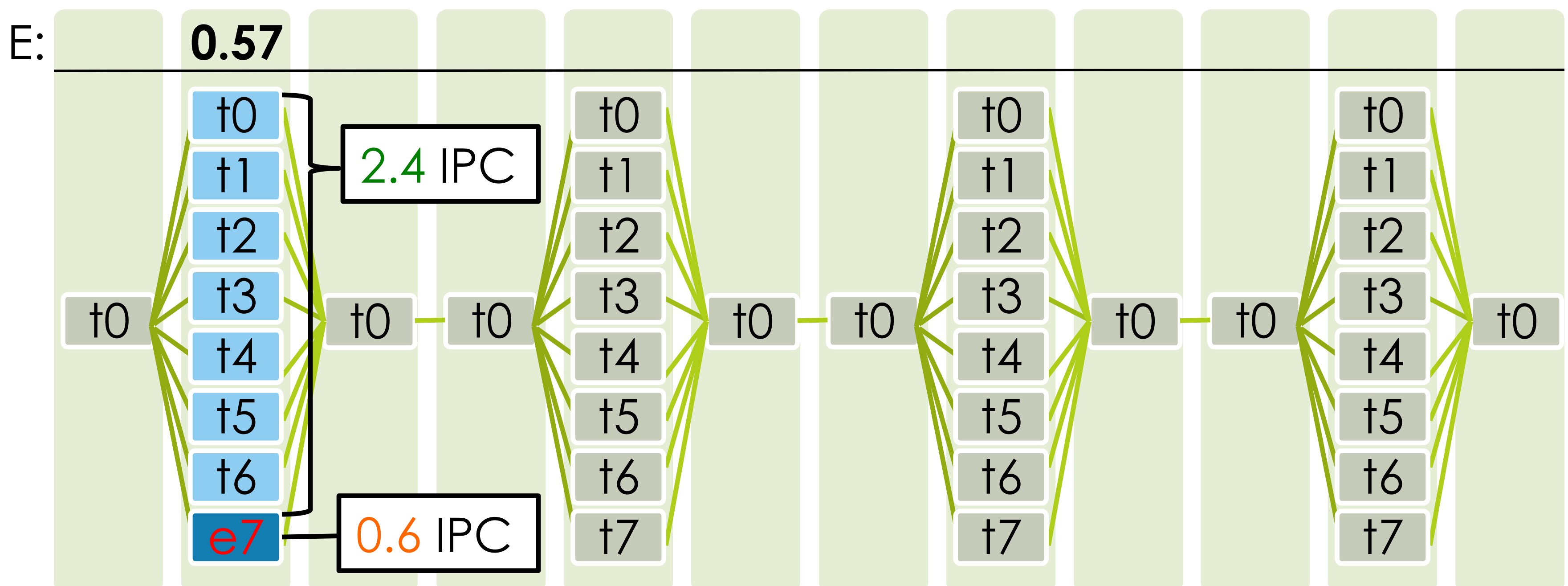
Performance results are recorded from the experiment and parallel section



Serial Experiments

Example of serial experiment in process **A**:

Parallel section's efficiency is estimated as
(**2.4**/**0.6**) / (**8**-**1**) = **0.57**

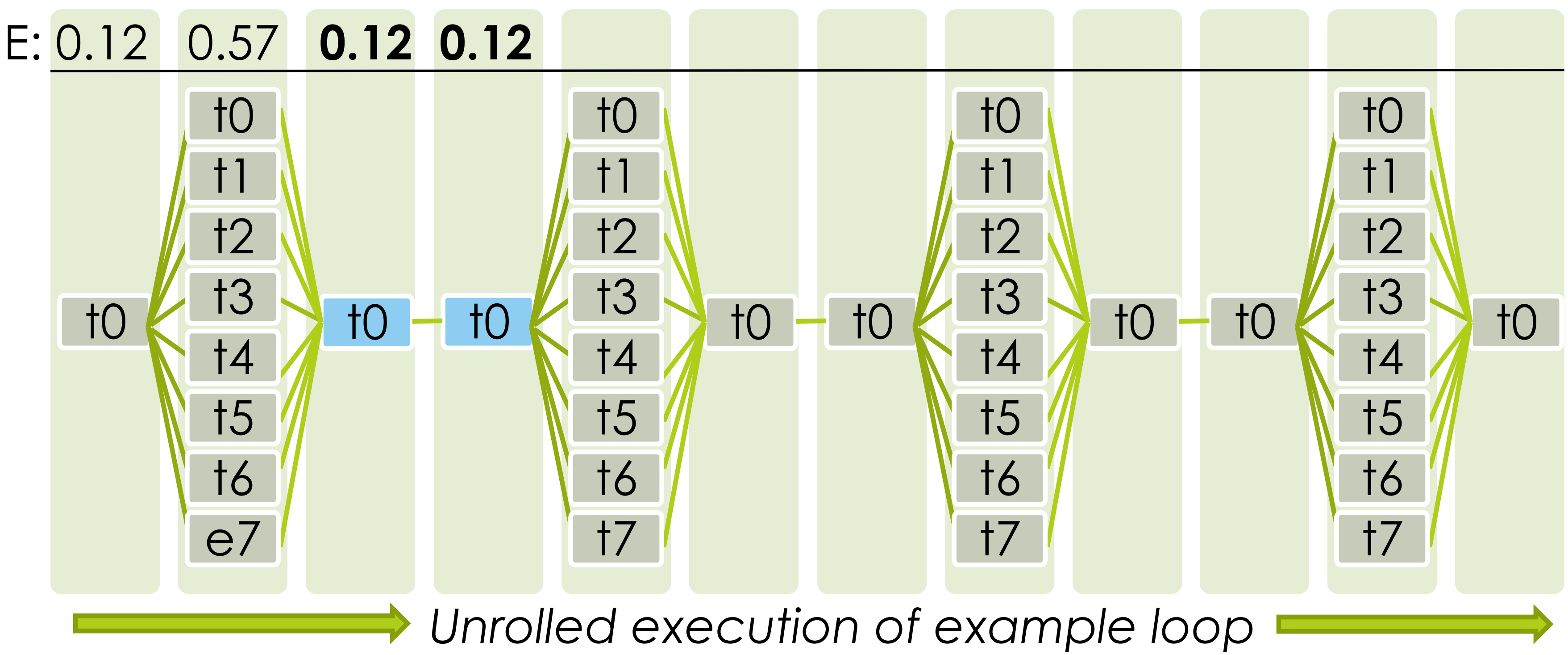


Unrolled execution of example loop

Serial Experiments

Example of serial experiment in process **A**:

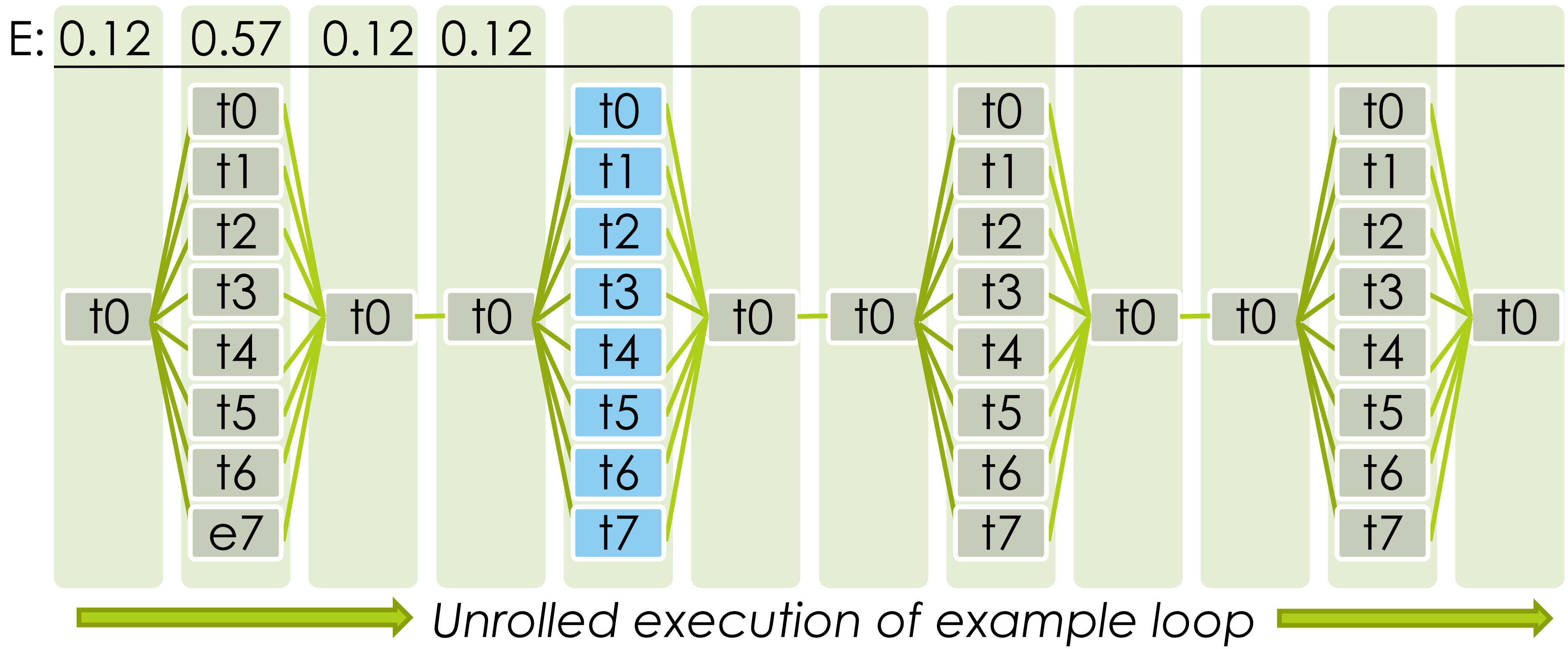
Serial phases proceed normally, with speedup of 1, efficiency 1/**8** = **0.125**



Serial Experiments

Example of serial experiment in process **A**:

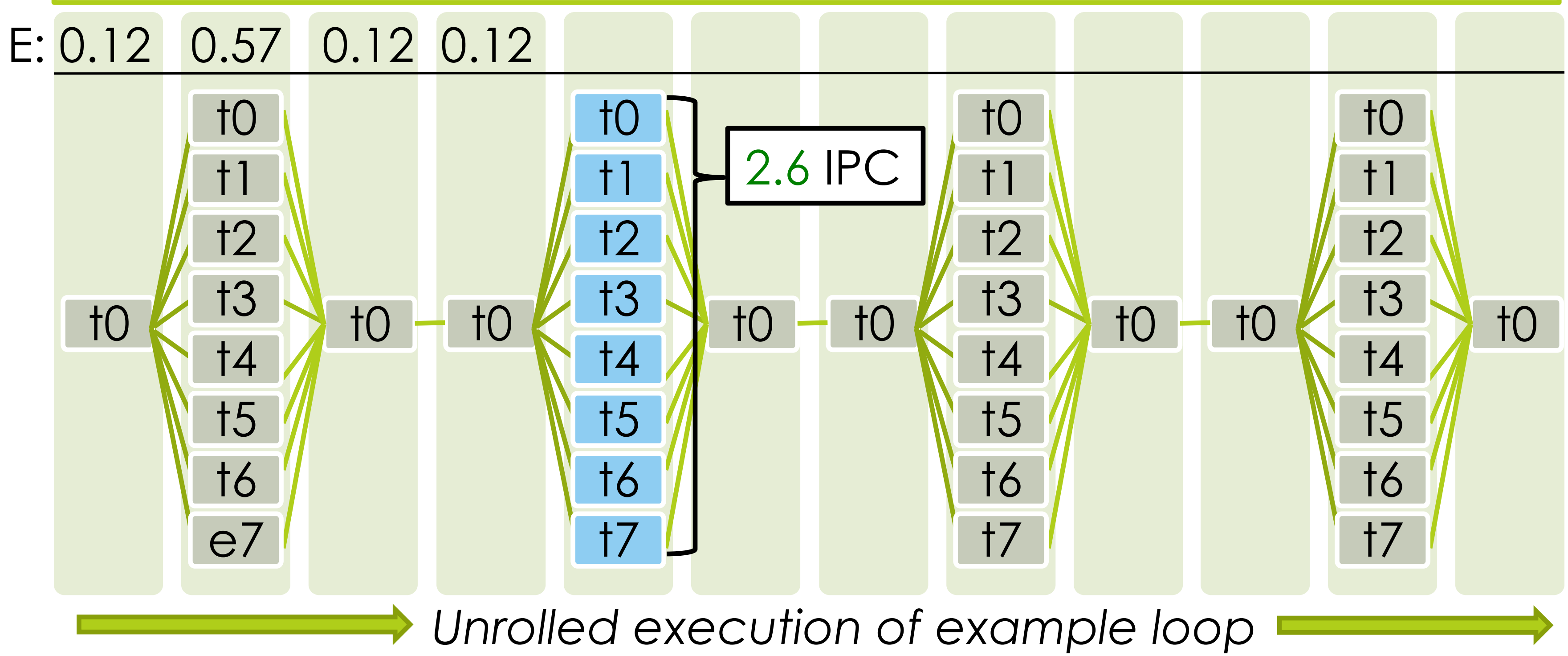
Same parallel section: no need to re-run the serial experiment



Serial Experiments

Example of serial experiment in process **A**:

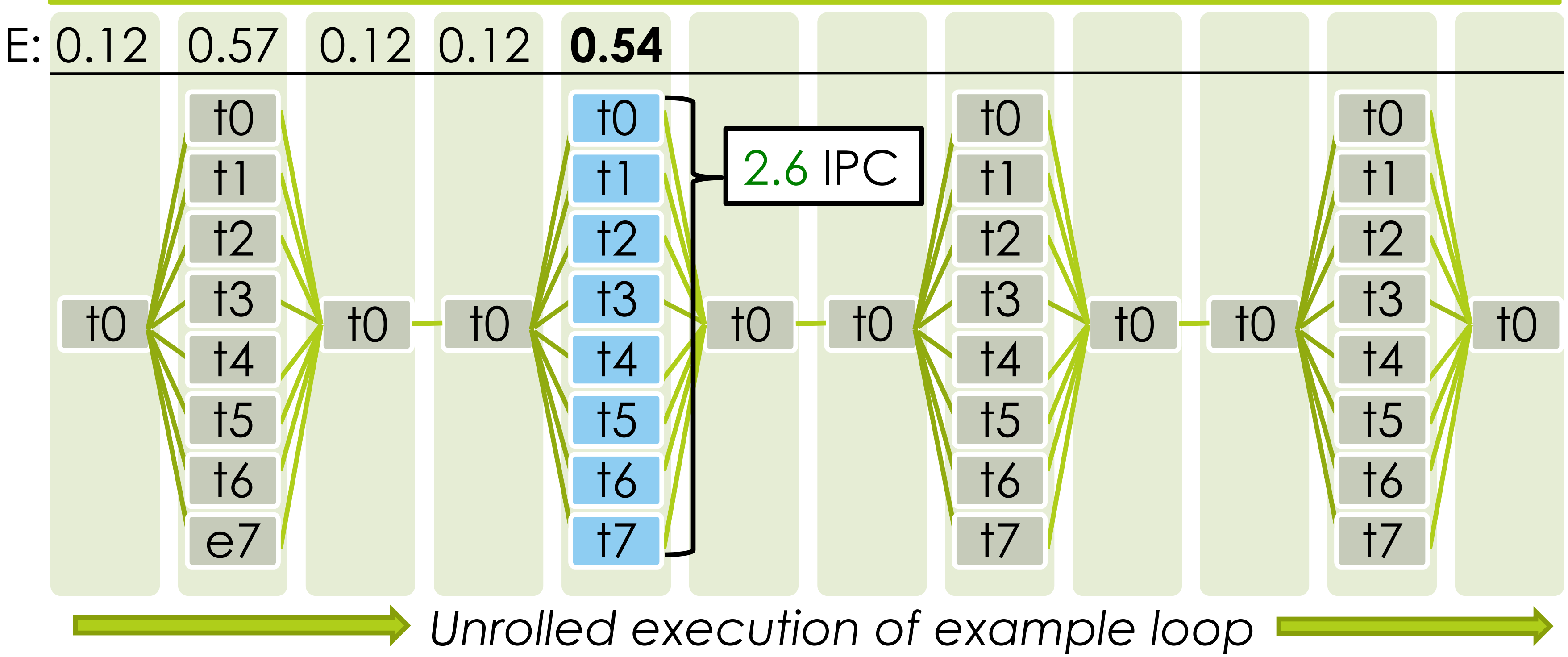
The latest achieved parallel performance is recorded



Serial Experiments

Example of serial experiment in process **A**:

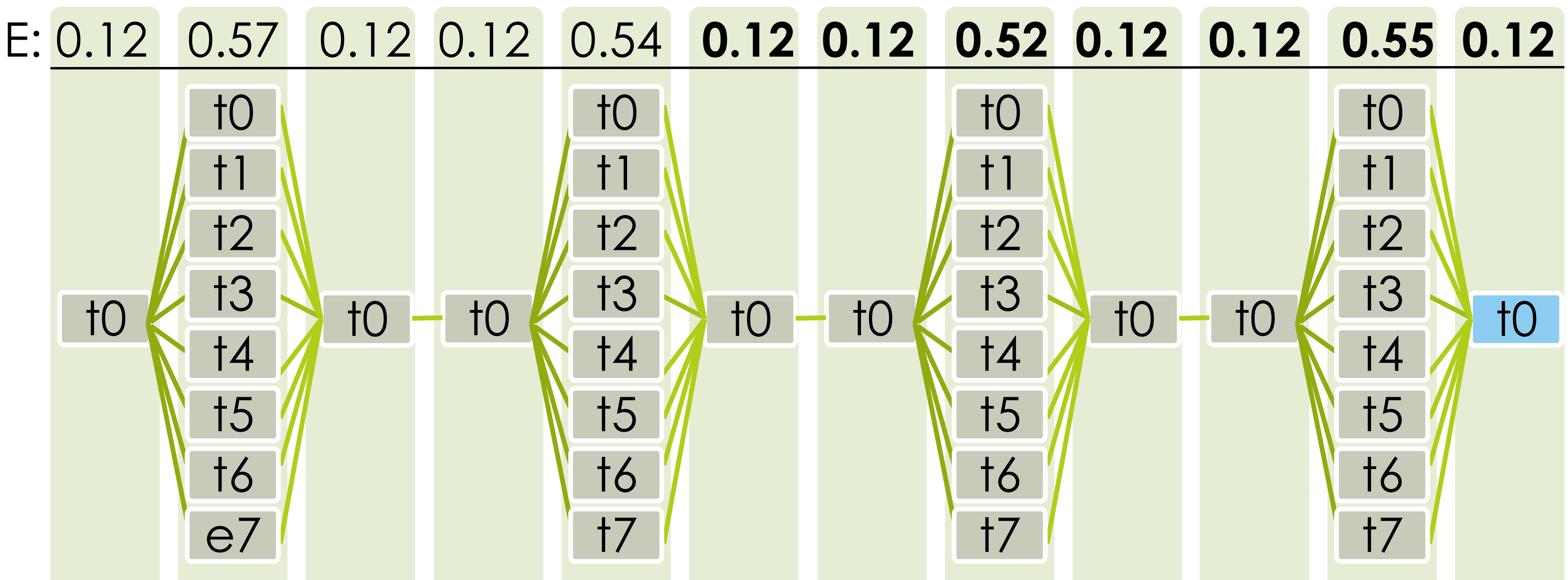
Latest efficiency for this section estimated at (**2.6**/**0.6**) / **8** = **0.54**



Serial Experiments

Example of serial experiment in process **A**:

And so on!

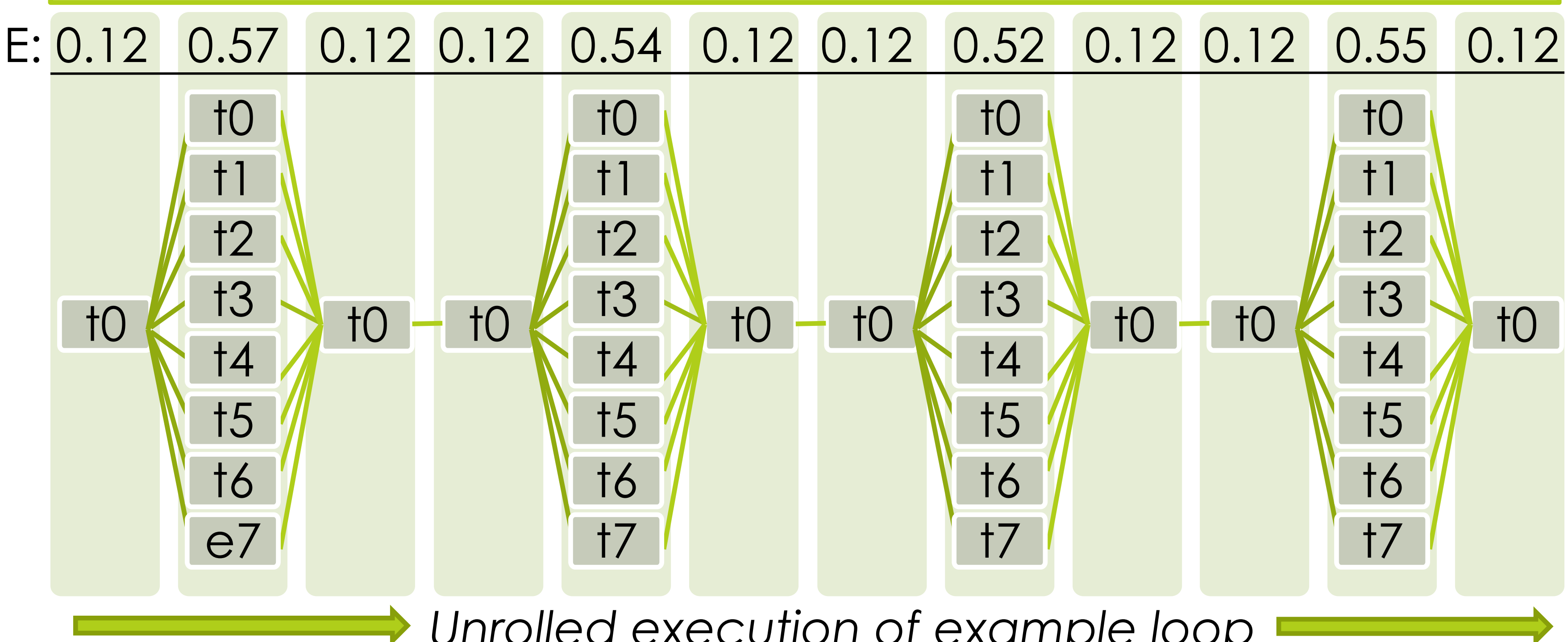


Unrolled execution of example loop

Serial Experiments

Example of serial experiment in process **A**:

Efficiencies are passed through a low-pass filter, accounting for durations. This "smooths" the alternating serial/parallel efficiencies present in applications.

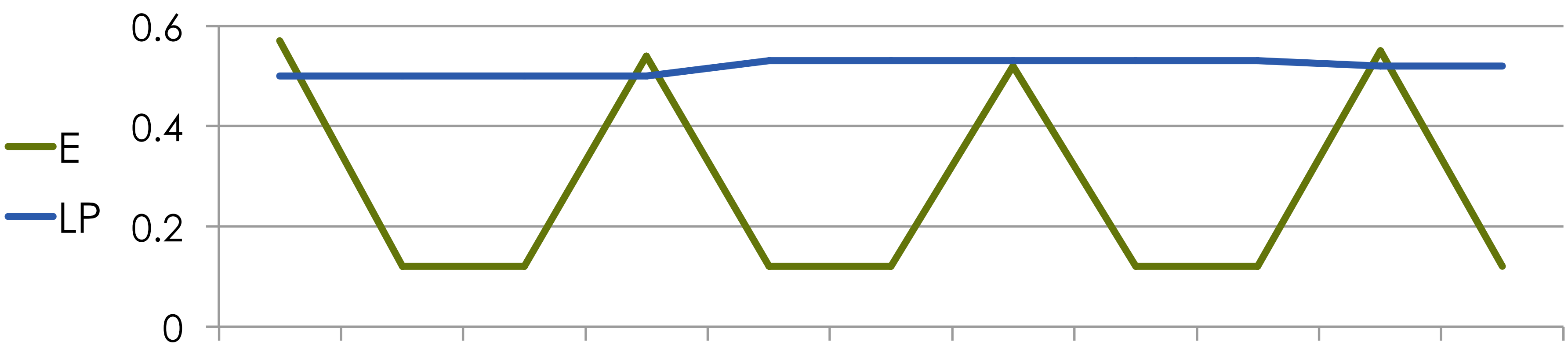


Serial Experiments

Example of serial experiment in process A:

Efficiencies are passed through a low-pass filter, accounting for durations. This "smooths" the alternating serial/parallel efficiencies present in applications.

E: 0.12 0.57 0.12 0.12 0.54 0.12 0.12 0.52 0.12 0.12 0.55 0.12



***Low-pass filter**: attenuates high-frequency signal content*

Evaluation

- Summary
- Example Scenario

Summary

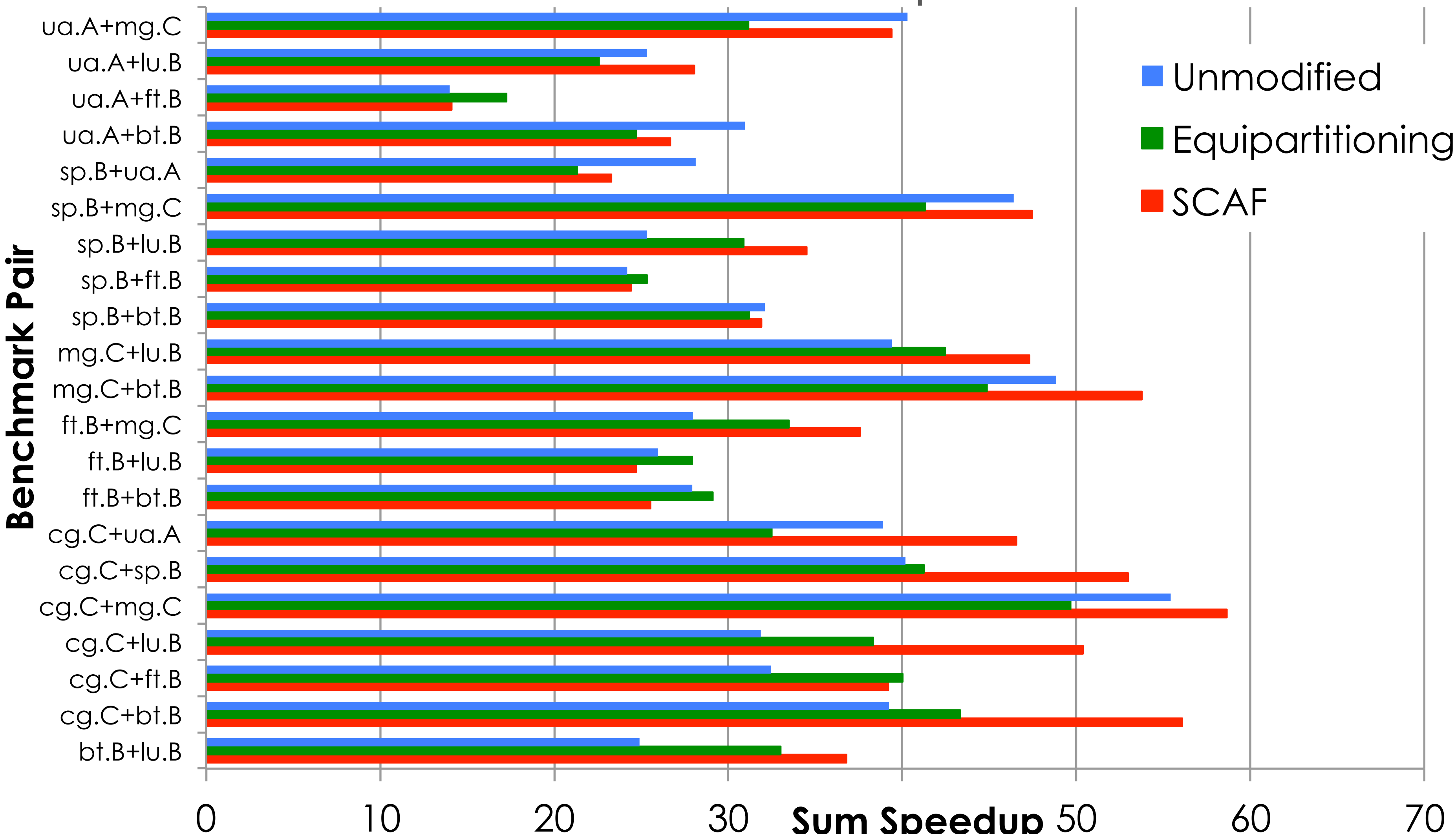
- Pairwise (2-way) multiprogramming of NAS benchmarks
  - 57% (Sparc T2) and 70% (Xeon) of benchmarks see improvement over **both** equipartitioning and oversubscription
  - Mean improvement was 15% increase in sum of both speedups over fastest

Summary

- Pairwise (2-way) multiprogramming of NAS benchmarks
  - 57% (Sparc T2) and 70% (Xeon) of benchmarks see improvement over **both** equipartitioning and oversubscription
  - Mean improvement was 15% increase in sum of both speedups over fastest
  - Up to 57% (T2) and 307% (Xeon) improvement over oversubscription

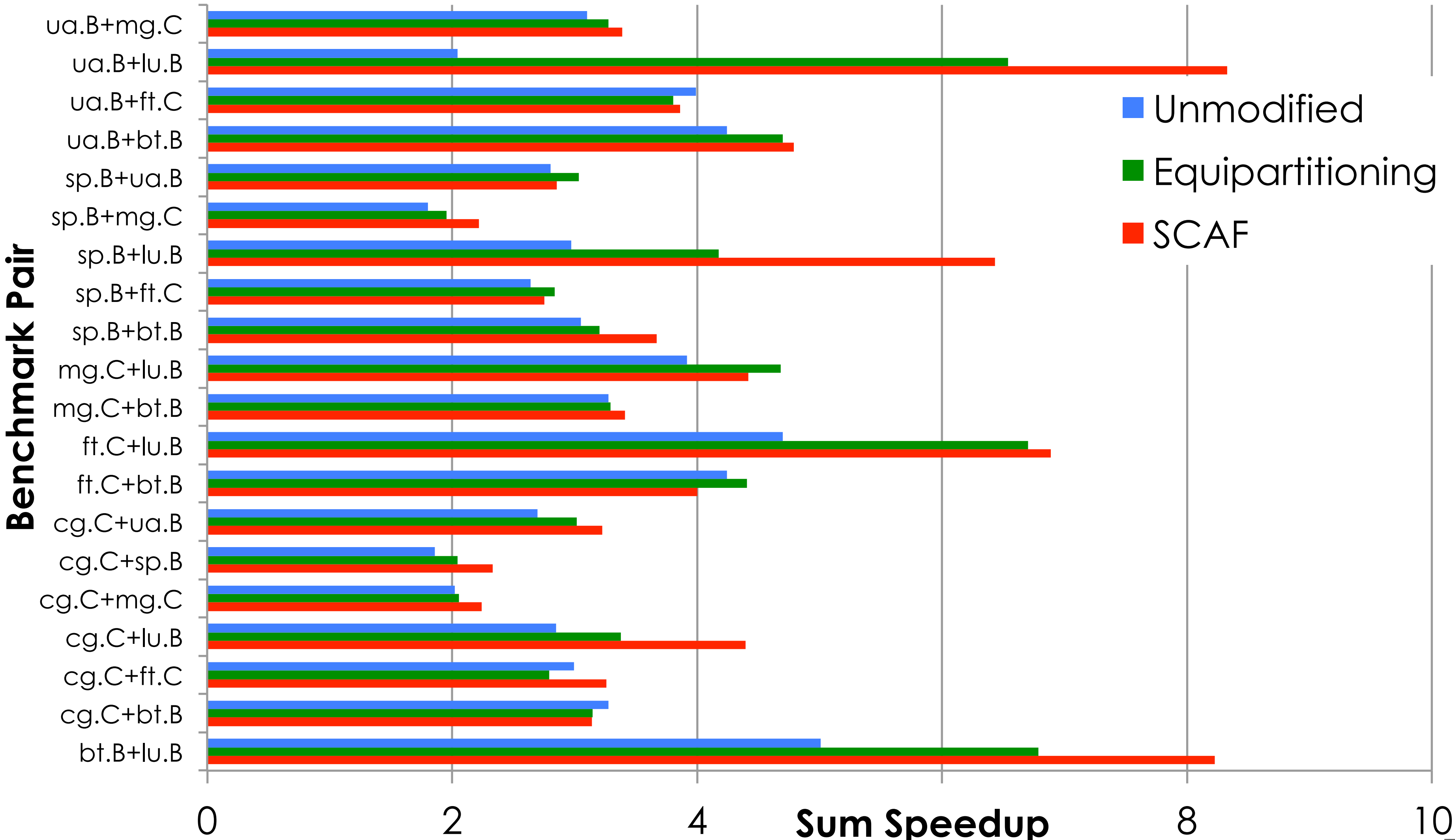
Summary

- Pairwise NAS results on Sparc T2



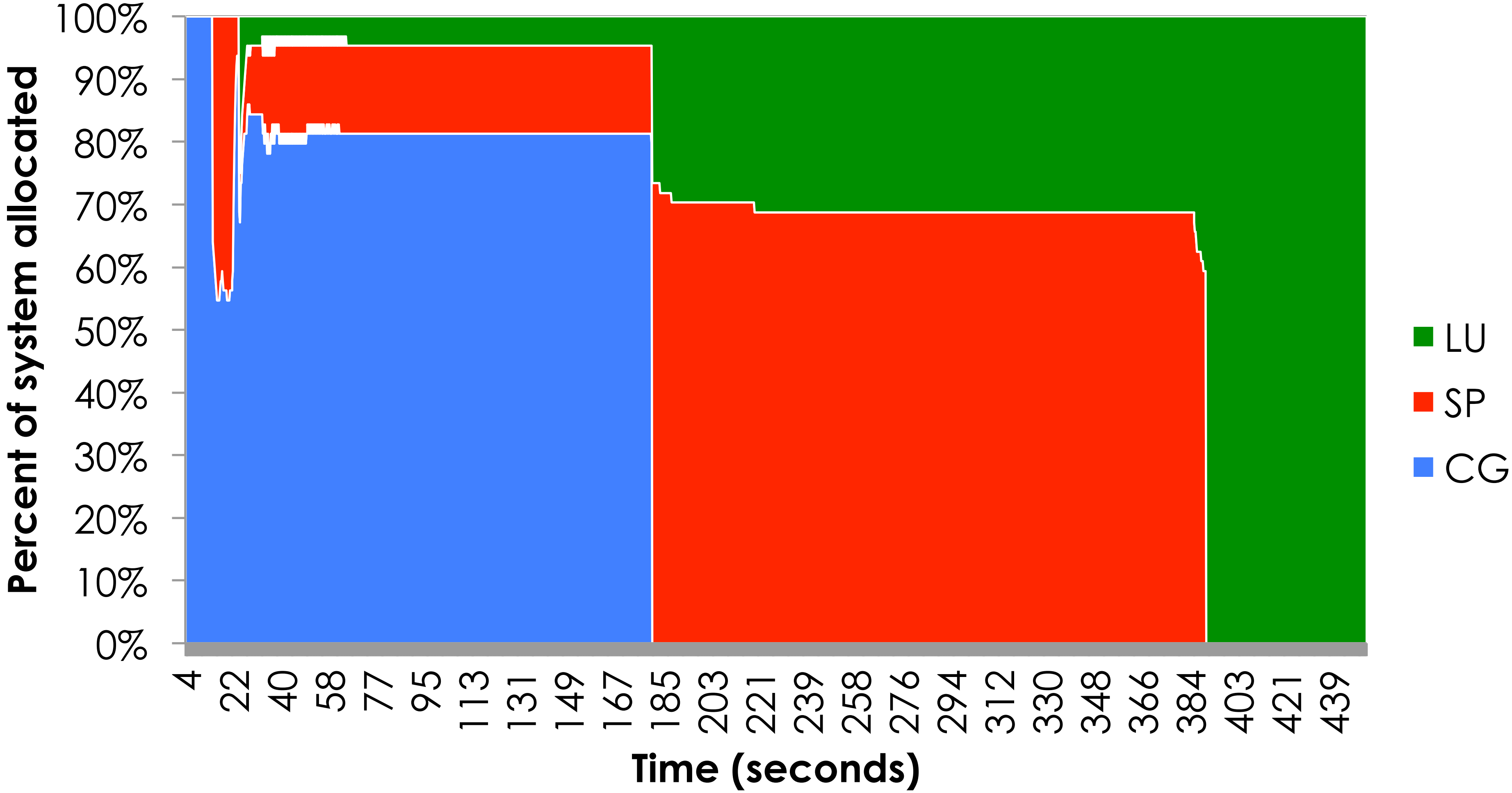
Summary

- Pairwise NAS results on 2x Xeon E5410



Example Scenario

- Detail of a 3-way multiprogramming scenario:



Example Scenario

- Detail of a 3-way multiprogramming scenario:
  - In this case, **45%** improvement vs. equipartitioning

Scheme	Process	Runtime	Speedup	Σ Speedup
Unmodified	CG SP LU	436s 475s 507s	13.1 9.6 8.8	31.5
Equi-partitioning	CG SP LU	374s 381s 350s	15.5 12.2 13.0	40.7
SCAF	CG SP LU	172s 374s 424s	35.7 12.5 11.1	**59.3**

Questions?