AfterOMPT: An OMPT-based tool for fine-grained tracing of tasks and loops

Igor Wodiany, Andi Drebes, Richard Neill, Antoniu Pop

International Workshop on OpenMP 2020

Introduction

- Need for precise profiling to identify performance anomalies
- OMPT allows for the implementation of portable profiling tools for OpenMP applications:
  - Few OMPT-based tools available
  - OMPT provides only limited information on loops
- Existing OpenMP profiling tools:
  - non-portable across run-times (e.g. Intel VTune)
  - no precise information on loops (e.g. Score-P)
  - not suitable for certain analysis (e.g. Grain Graphs)

OMPT

- OMPT defines a set of callbacks signatures and declarations, e.g.

```
typedef void (*ompt_callback_thread_begin_t) (
  ompt_thread_t thread_type,
  ompt_data_t* thread_data);

typedef void (*ompt_callback_task_schedule_t) (
  ompt_data_t*prior_task_data,
  ompt_task_status_tprior_task_status,
  ompt_data_t*next_task_data);
```

OpenMP 5.0 Specification https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

- It allows for external tools to link custom code to each callback, to be invoked by the run-time at execution-time

OMPT Loop Tracing is Limited

- Currently only supported via the generic callback `ompt_callback_work`, simply dispatched at start and end of the loop
- Misses important information specific to the loop and its loop chunks:
  - The loop's iteration space
  - Partitioning of the iteration space into chunks
  - Mapping of those chunks onto CPUs
  - The execution interval of each chunk
- Extension to OMPT proposed before [1]

[1] Langdal, P.V., Jahre, M., Muddukrishna, A.: Extending OMPT to support grain graphs. In: International Workshop on OpenMP. pp. 141–155. Springer (2017)

Our Contributions

- AfterOMPT – Aftermath-based profiling tool that implements the OMPT interface
- Implementation of the OMPT extension for loop tracing
- Two case studies supporting extension of the OMPT interface
- Overhead analysis of our profiling tool

AfterOMPT

- Implements OMPT interface
- Uses Aftermath tracing API for data collection
- Enables tracing of loops, tasks and synchronization events

Aftermath

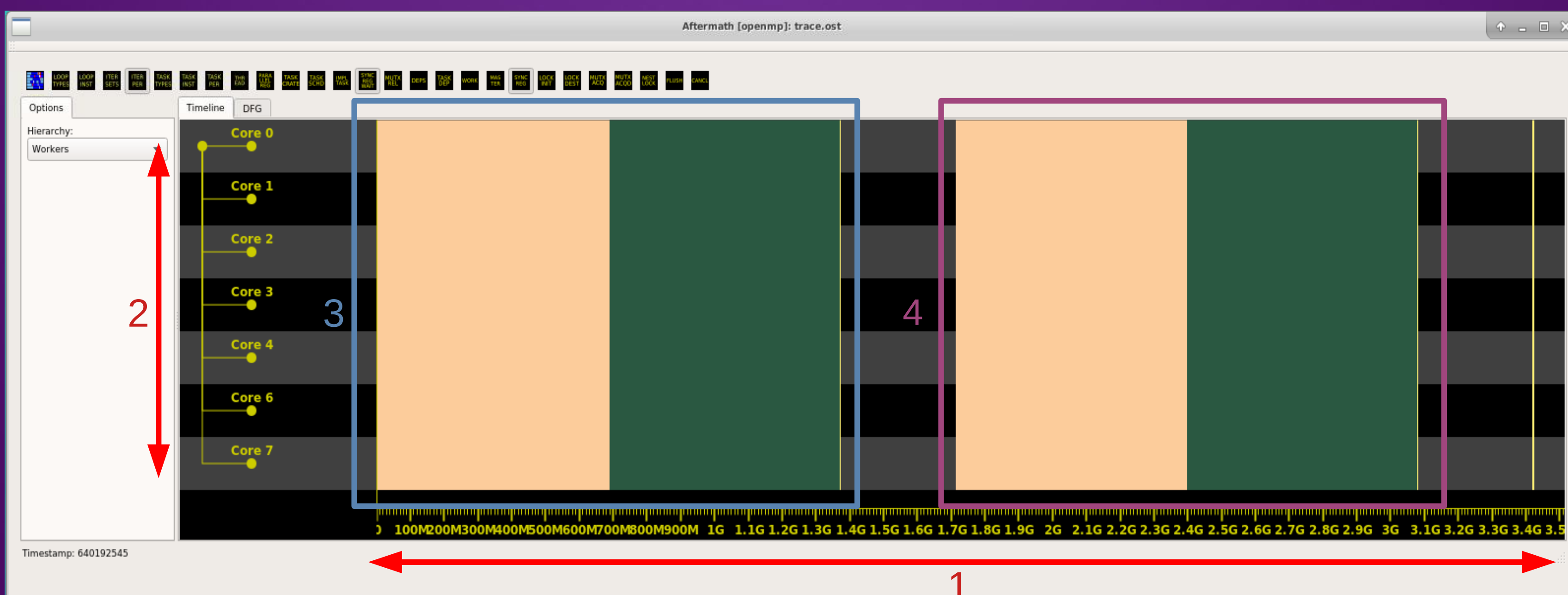
- Tracing and visualization tool for performance analysis
- OpenMP previously supported, but not portable, as an instrumented run-time was required
- New version extended to represent OMPT events
- Available for free: https://www.aftermath-tracing.com/

Aftermath

1. Timeline
2. CPU Cores
3. Static Loop
4. Dynamic Loop

```
#pragma omp parallel num_threads(8)
{
  #pragma omp for schedule(static, 2) // First loop
  for(int i = 0; i < 32; i++) { foo(); }
  foo();
  #pragma omp for schedule(dynamic, 2) // Second loop
  for(int i = 0; i < 32; i++) { foo(); }
  foo();
}
```

Each loop allocates 4 iterations per worker = 2 loop chunks



Proposed OMPT Extension

- Enable more detailed and fine-grained (chunk-level) tracing of OpenMP loops
- Based on the previous proposal by Langdal et al., however:
  - We use *_begin and *_end callbacks
  - We do not include the chunk creation time and the last chunk marker
- Proof-of-concept implemented in LLVM 9.0 run-time and in our tool
- Static loop tracing may require modification of the compiler

Loop Callback

Proposed Extension

```
typedef void (*ompt_callback_loop_begin_t) (
  ompt_data_t* parallel_data,
  ompt_data_t* task_data,
  int flags,
  int64_t lower_bound,
  int64_t upper_bound,
  int64_t increment,
  int num_workers,
  void* codeptr_ra);

typedef void (*ompt_callback_loop_end_t) (
  ompt_data_t* parallel_data,
  ompt_data_t* task_data);
```

Loop Chunk Callback

Proposed Extension

```
typedef void (*ompt_callback_loop_chunk_t) (
  ompt_data_t* parallel_data,
  ompt_data_t* task_data,
  int64_t lower_bound,
  int64_t upper_bound);
```

Case Studies

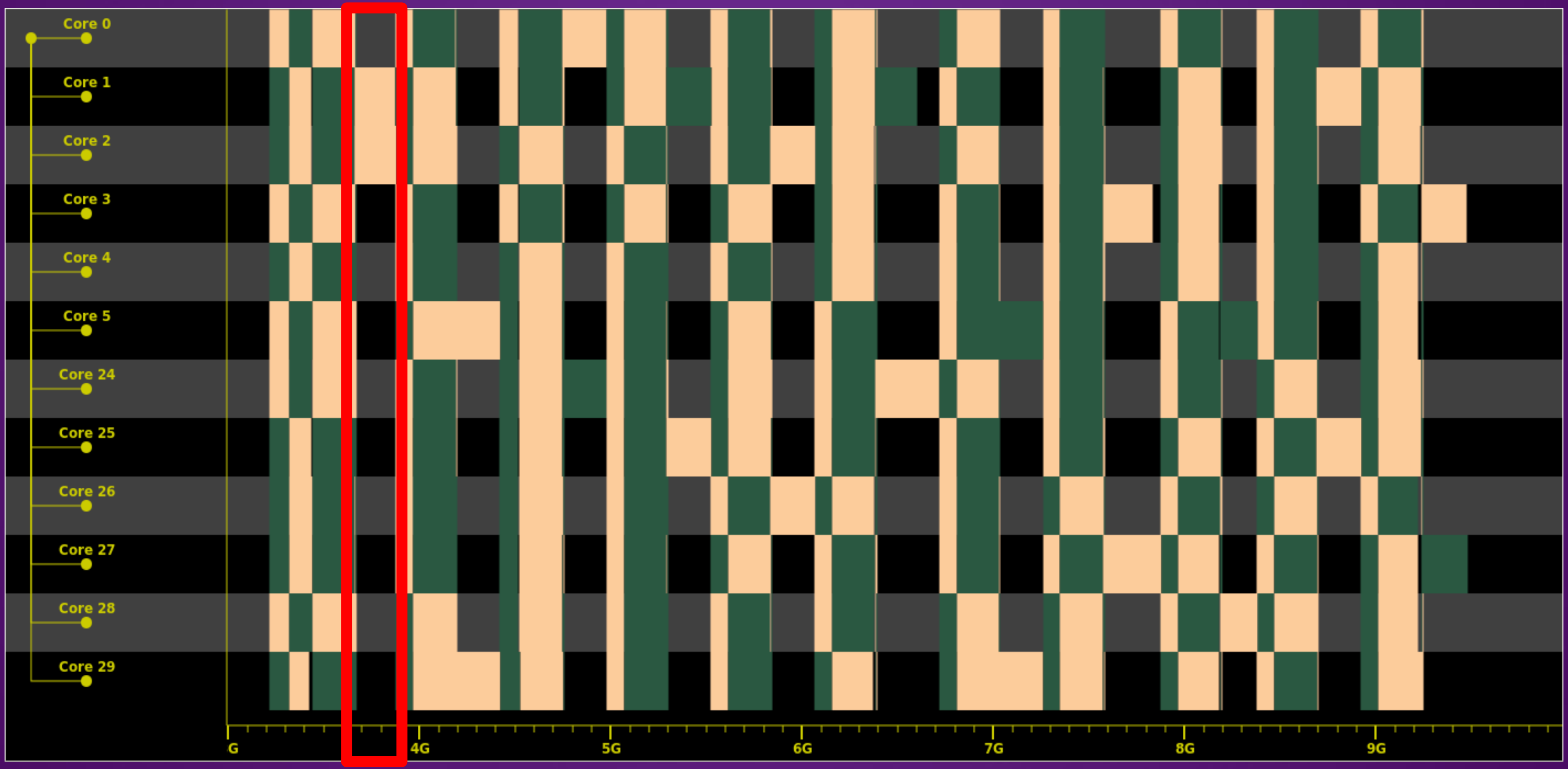
- Concrete examples where more precise loop tracing is needed
- Use cases focused on:
  - Helping less experienced developers
  - Making identification of performance anomalies easier

Case Study I: Imbalanced Loops

- *IS* benchmark from NPB
- Loop-based integer bucket sort
- Range of the input data changed to cause an underutilization of some of the buckets

Case Study I: Imbalanced Loops

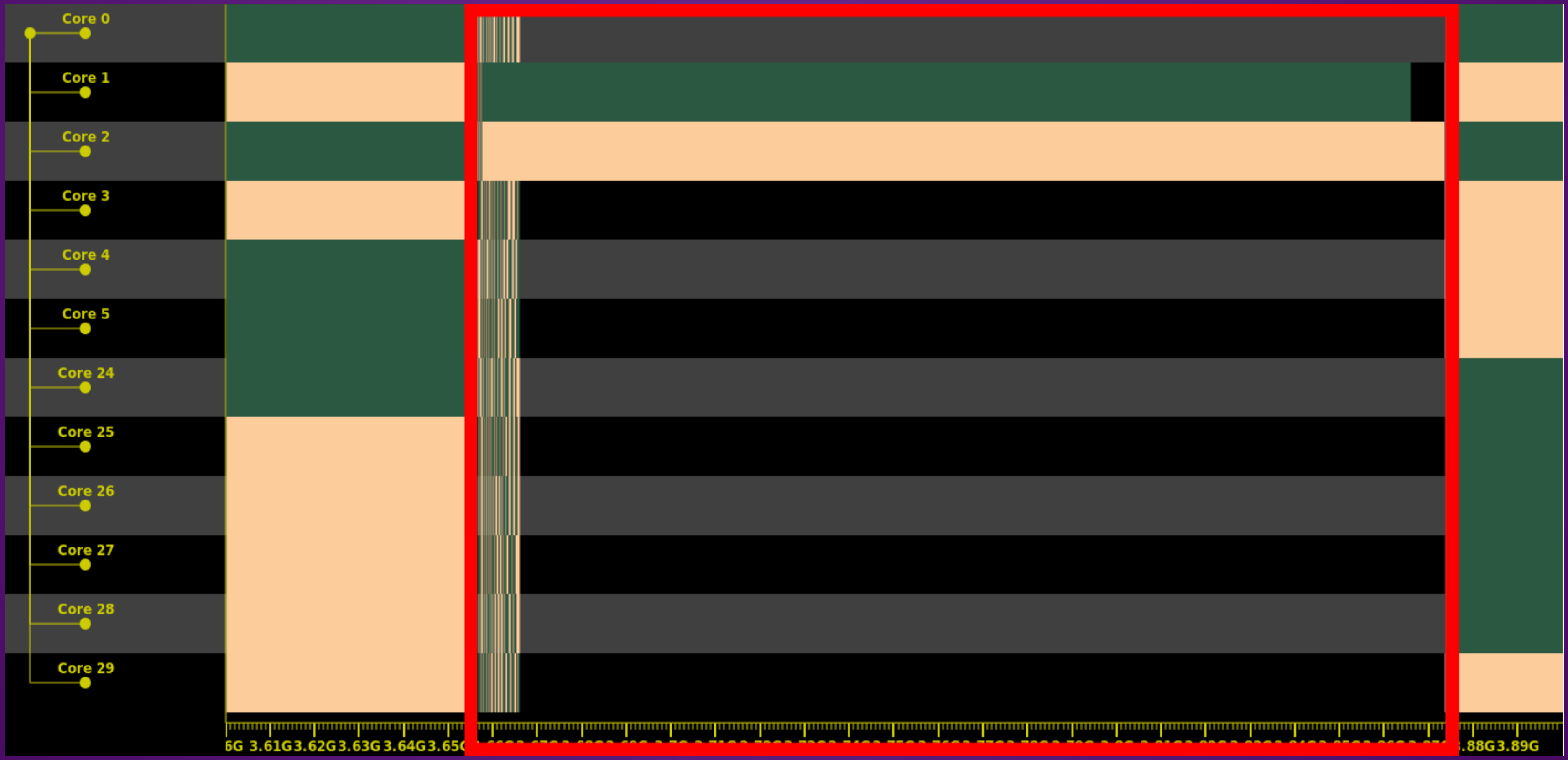
Execution of the full application



IS from NPB

Case Study I: Imbalanced Loops

Execution of one loop instance



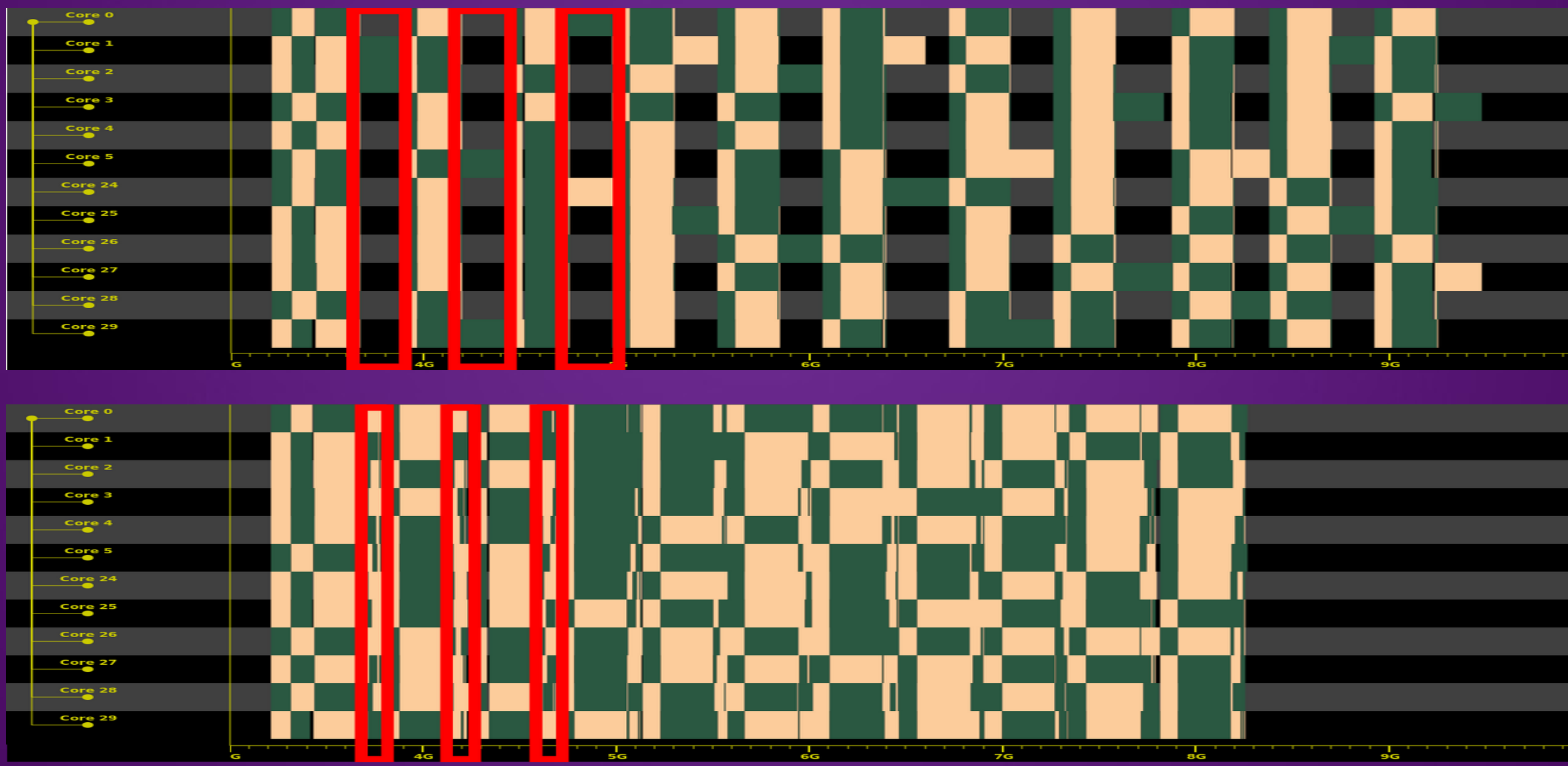
IS from NPB

Case Study I: Imbalanced Loops

- Tracing of loop chunks allows to identify anomalous iterations
- This lead to an easy identification of “overflowing” buckets
- 4x more buckets = 1.22x speed-up
- Could be done without the new callback, but extension makes it easy to pinpoint the problem

Case Study I: Imbalanced Loops

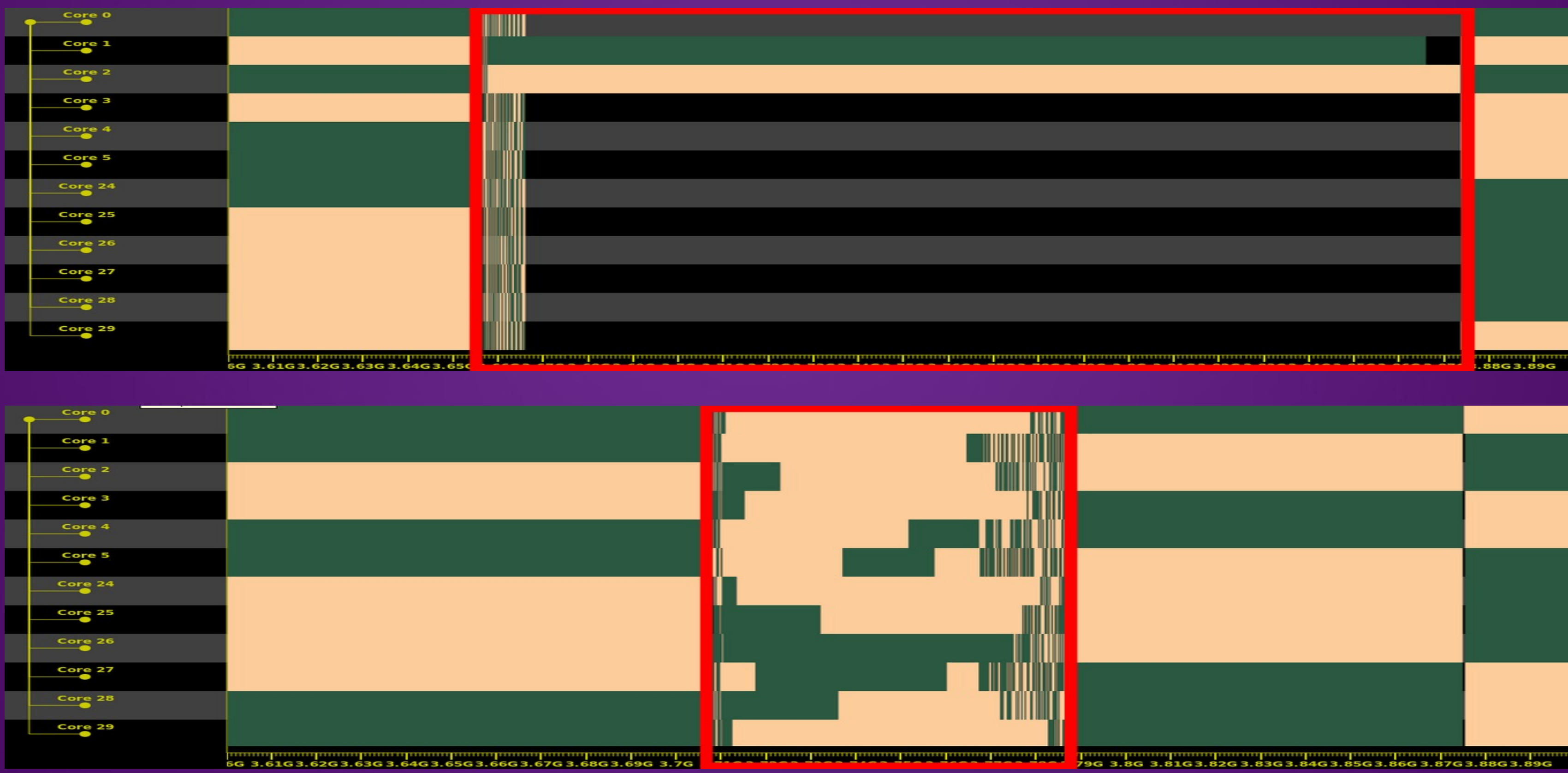
Initial code (top) and optimized version (bottom) – full application



IS from NPB

Case Study I: Imbalanced Loops

Initial code (top) and optimized version (bottom) – one loop



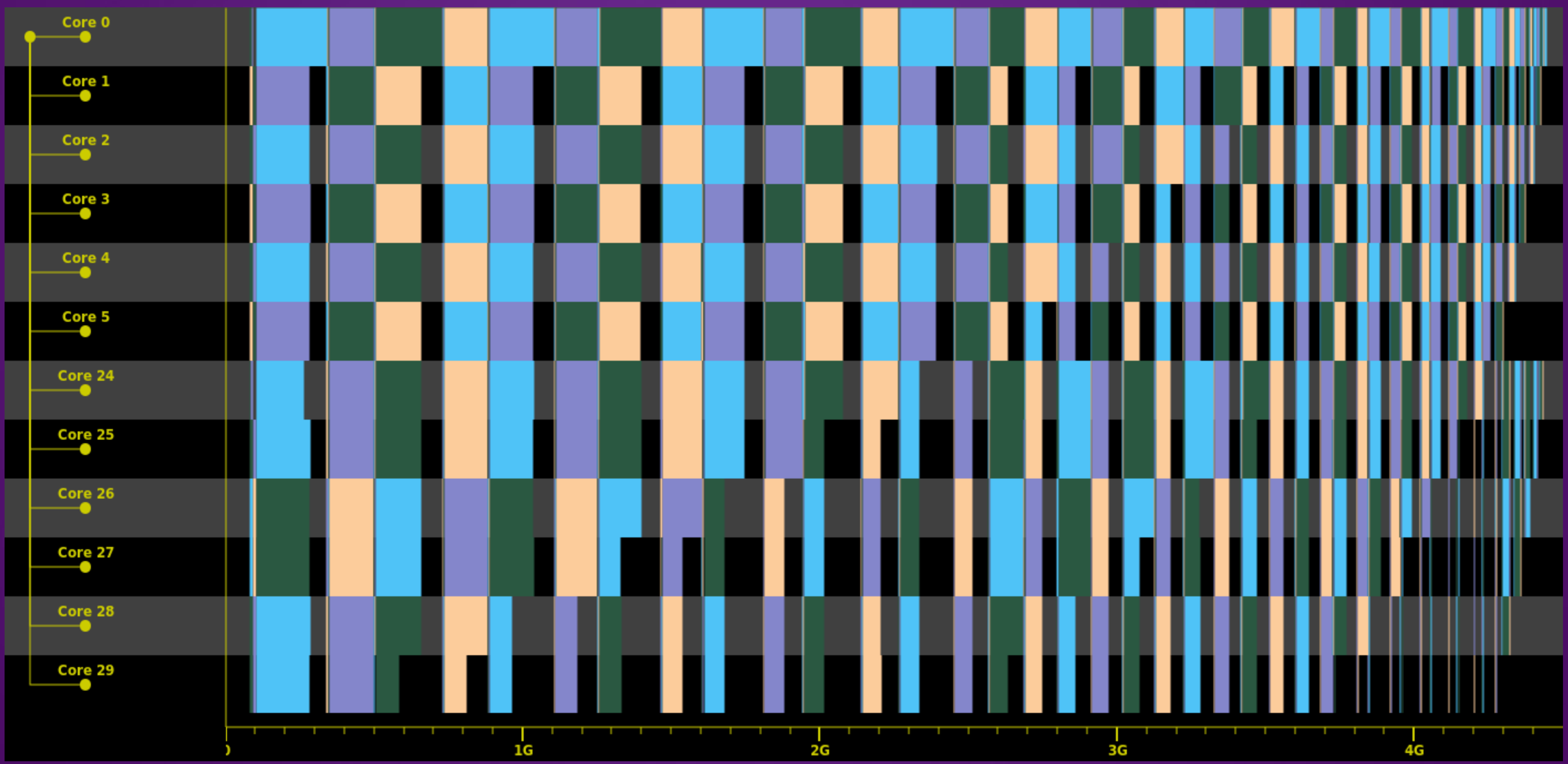
IS from NPB

Case Study II: Loops vs Tasks

- Help the programmer choose the parallel primitives with the best performance
- *SparseLU* benchmark from BOTS:
  - Three implementations: task-based and loop-based (static scheduling + dynamic scheduling)
  - Comparison of loop and task parallelism with AfterOMPT

Case Study II: Loops vs Tasks

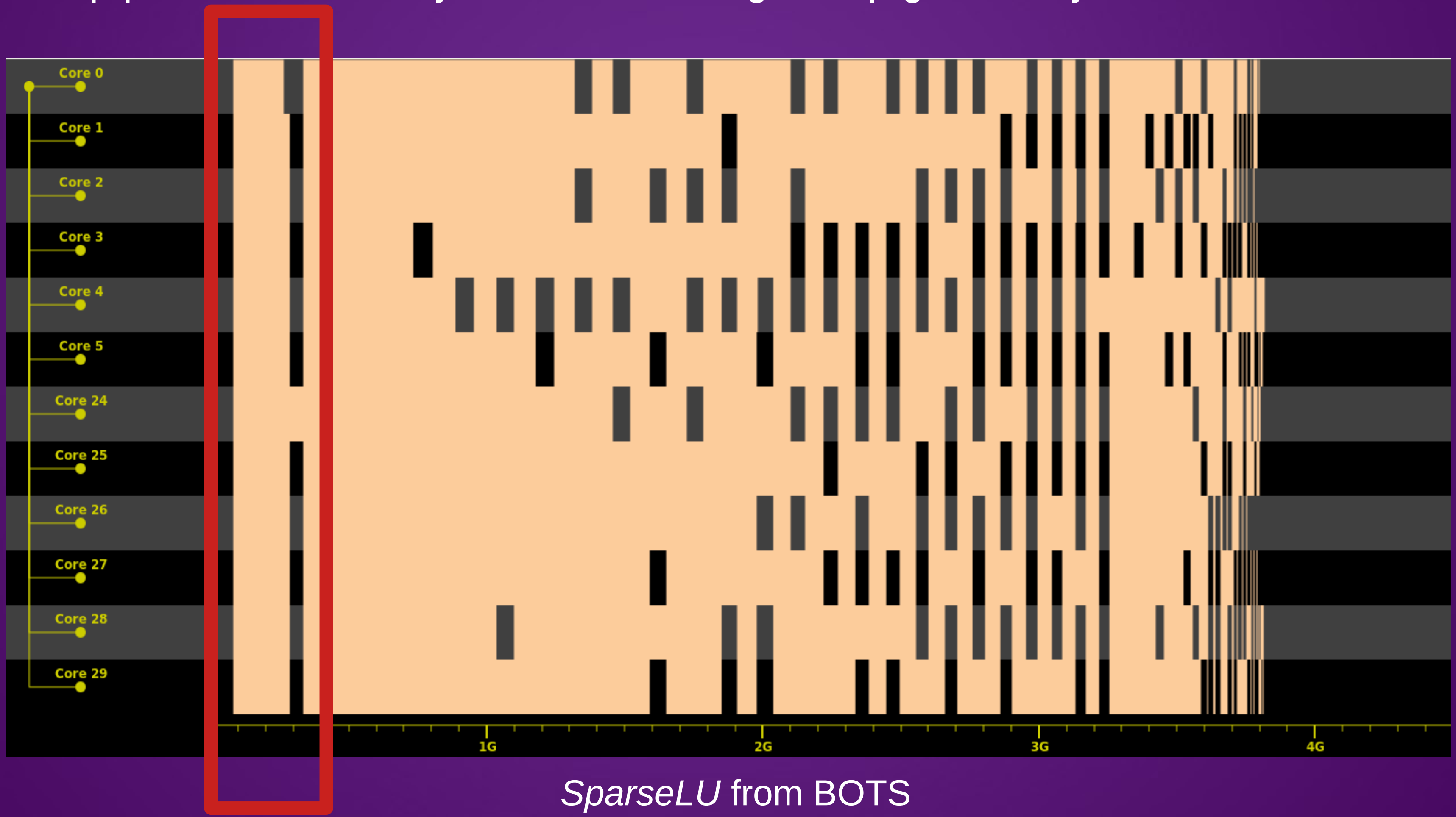
Loop parallelism with static scheduling



SparseLU from BOTS

Case Study II: Loops vs Tasks

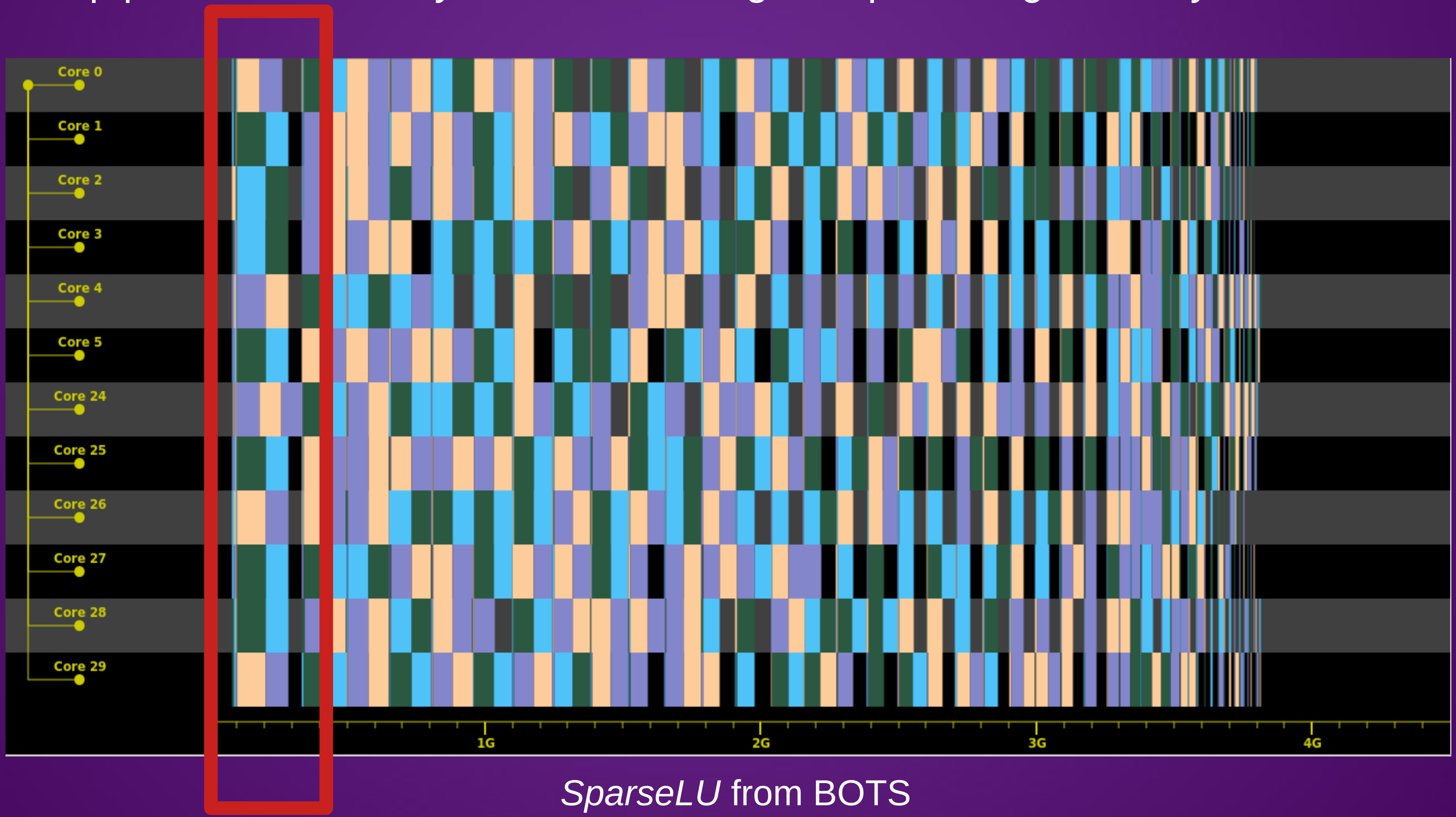
Loop parallelism with dynamic scheduling – loop granularity



SparseLU from BOTS

Case Study II: Loops vs Tasks

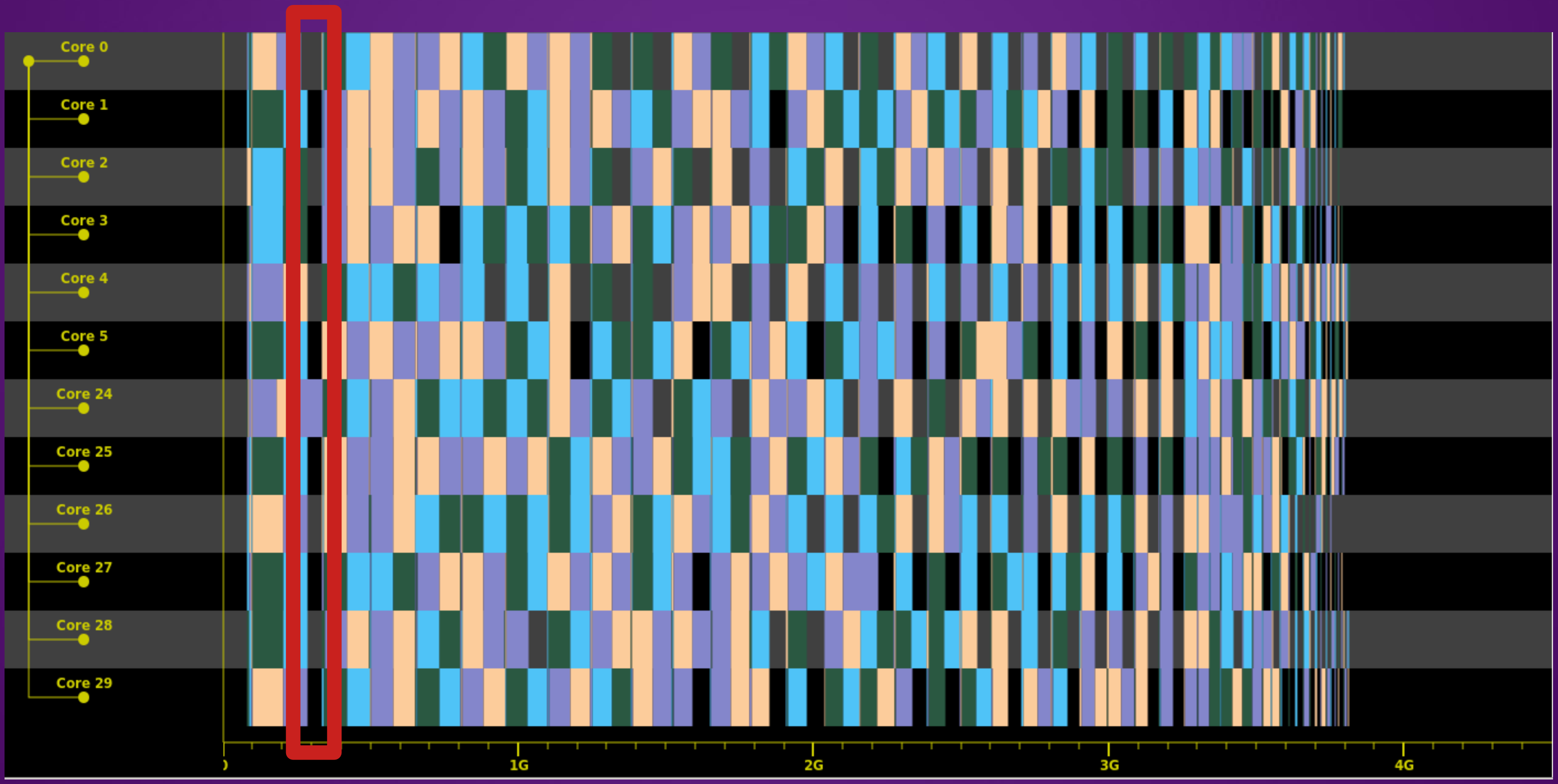
Loop parallelism with dynamic scheduling – loop chunk granularity



SparseLU from BOTS

Case Study II: Loops vs Tasks

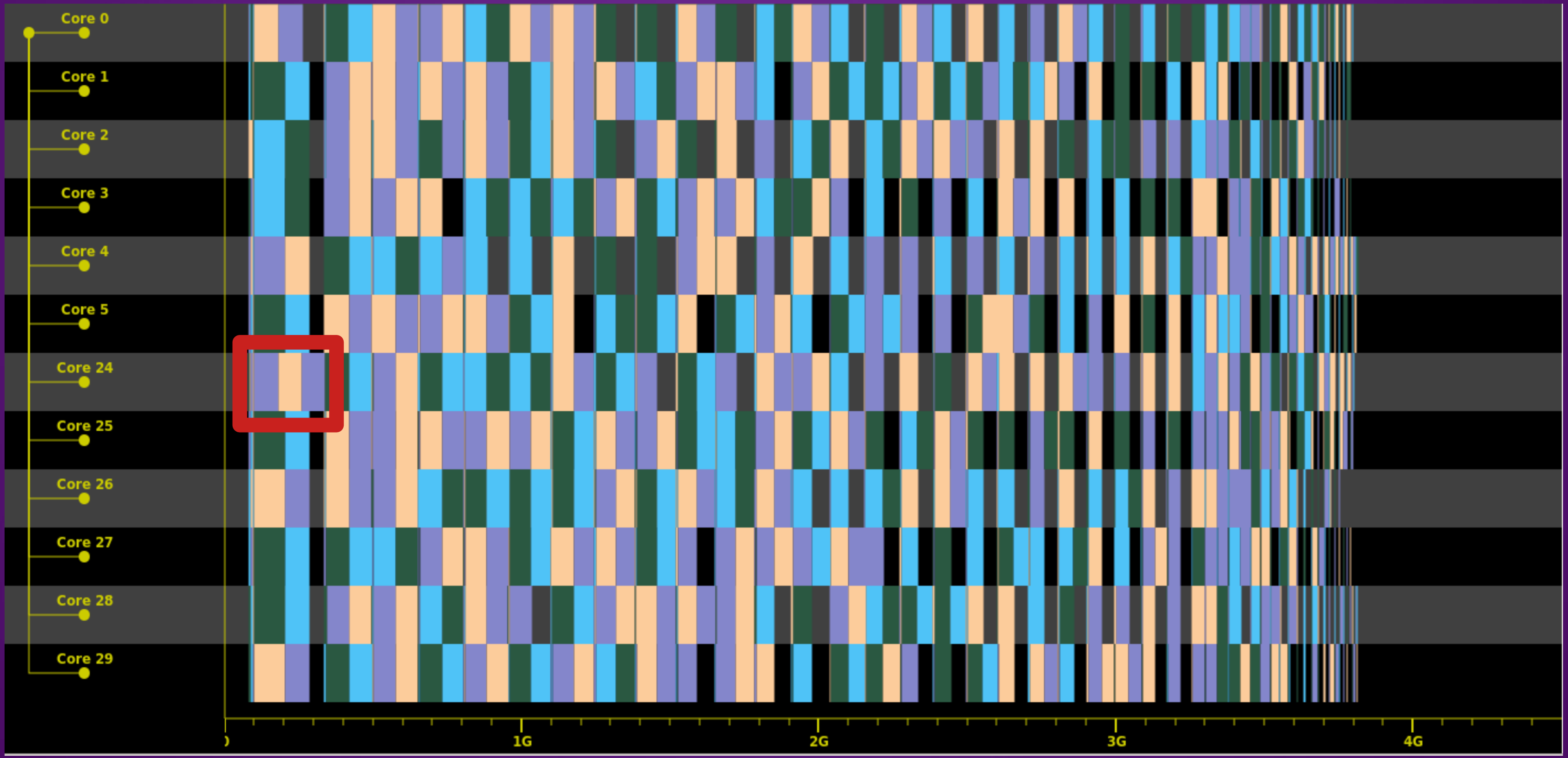
Loop parallelism with dynamic scheduling – loop chunk granularity



SparseLU from BOTS

Case Study II: Loops vs Tasks

Loop parallelism with dynamic scheduling – loop chunk granularity



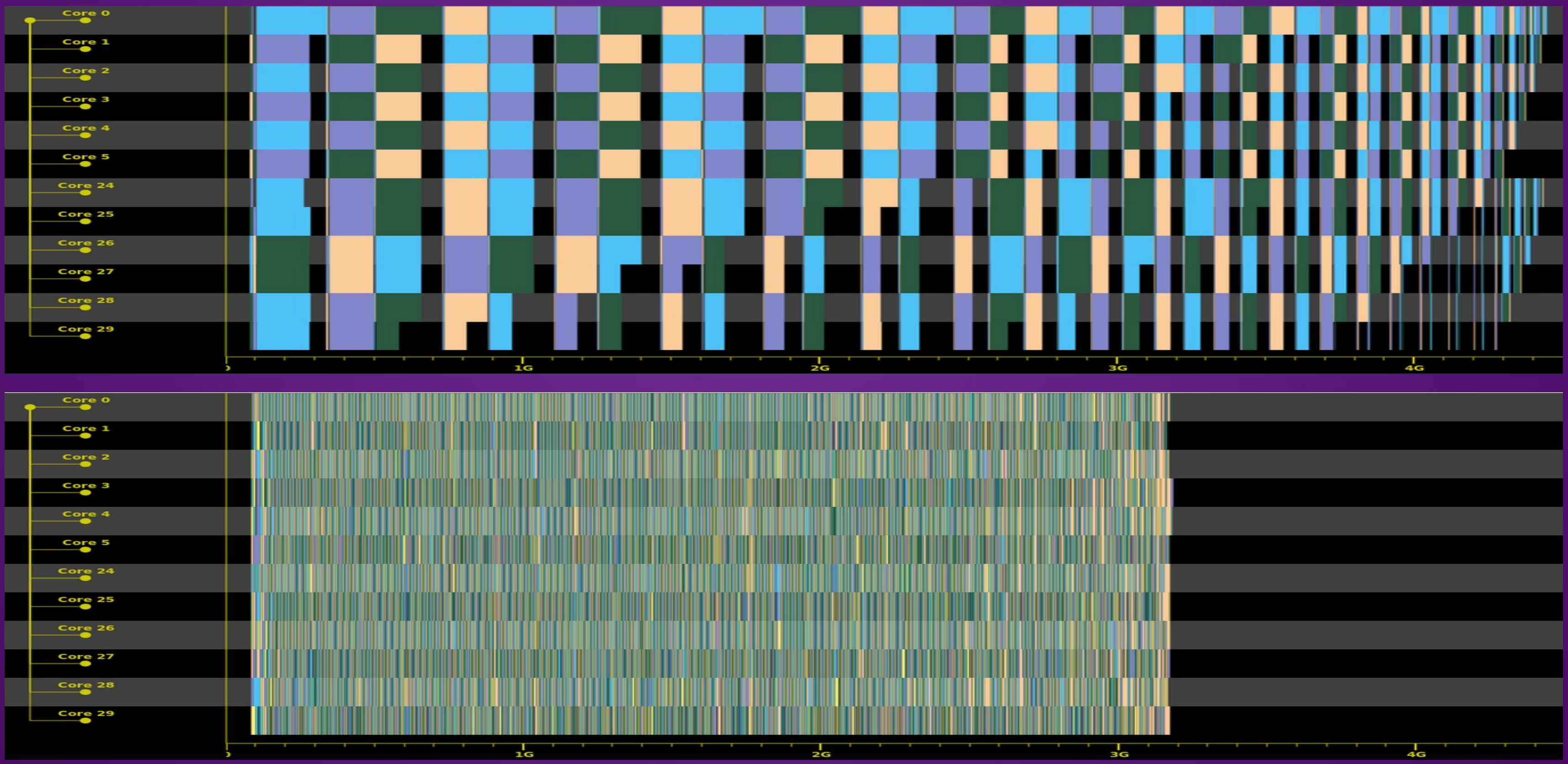
SparseLU from BOTS

Case Study II: Loops vs Tasks

- Per iteration work does not change
- So the problem is the work imbalance
- Uneven distribution of iterations is clearly visible
- Solutions:
  - Ensure #cores divides #iterations (what about performance portability?)
  - Introduce task-based parallelism
- This concludes cases studies on loop parallelism

Case Study II: Loops vs Tasks

Loop parallelism with static scheduling (top) and task parallelism (bottom)



SparseLU from BOTS

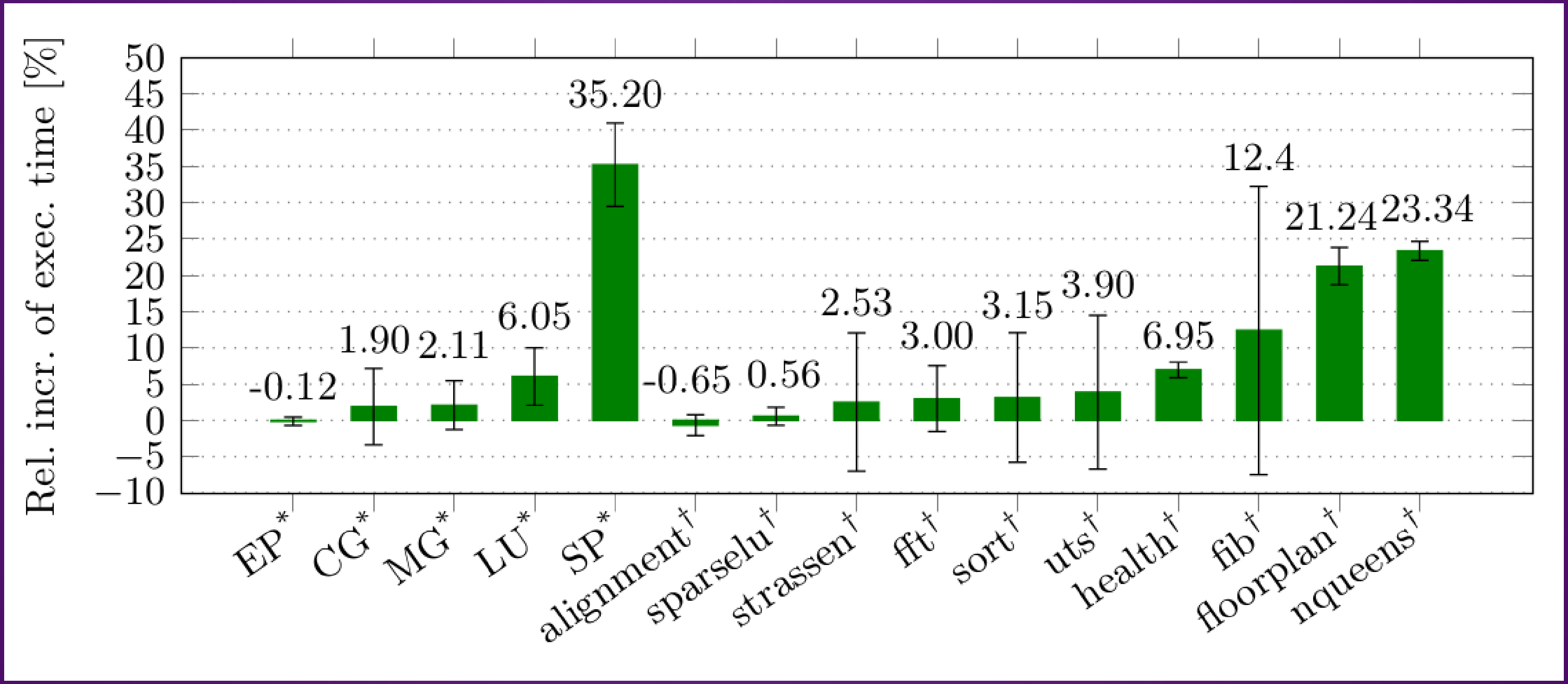
Overhead Analysis

- Tested on NPB* and BOTS** benchmarks
- Measured as an average relative increase of the execution time for 50 samples (0% = no overhead)
- Execution time measured as a wall clock time

* C implementation of NPB from https://github.com/benchmark-subsetting/NPB3.0-omp-C
** https://github.com/bsc-pm/bots

Overhead Analysis

(lower is better)



Overhead Analysis

- Overhead less than 5% for 9 out of 15 benchmarks
- Programs with small loop chunks (*LU*, *SP*) and small tasks (*fib*, *floorplan* and *nqueens*) incur a high overhead
- E.g., *floorplan*: ~10% of cycles spent in the task is an overhead (200 cycles overhead vs 2200 cycles work)
- Fixed high overhead and equal work per task can be acceptable

Conclusion

- Proposed an OMPT extension with new callbacks for precise and fine-grained loop tracing; and motivating use cases
- Presented AfterOMPT, an OMPT-based tool for fine-grained tracing of tasks and loops that implements the proposed extension
- Future work: hardware events profiling and task graph visualization
- GitHub: https://github.com/IgWod/ompt-loops-tracing
- Any questions? igor.wodiany@manchester.ac.uk

MANCHESTER
1824

The University of Manchester