Task Analysis with Score-P

27.06.2012 |

Daniel Lorenz
Jülich Supercomputing Centre

CScADS Tools Workshop
Snowbird, Utah

Overview

- Introduction to Score-P
- Task related performance issues
- Reconciling tasking with existing techniques
- Example analysis
- Future work

Score-P

Introduction to Score-P

- Common tools infrastructure:
  - Starting with Scalasca, Periscope, TAU, and Vampir
  - Open for other tools and groups
- SILC and LMAC projects funded by BMBF, Germany
  - Scalable Infrastructure for Automatic Performance Analysis of Parallel Codes (SILC)
  - Performance dynamics of massively parallel codes (LMAC)
- PRIMA project funded by DOE, US
  - Performance Refactoring of Instrumentation, Measurement, and Analysis Technolgies for Petascale Computing

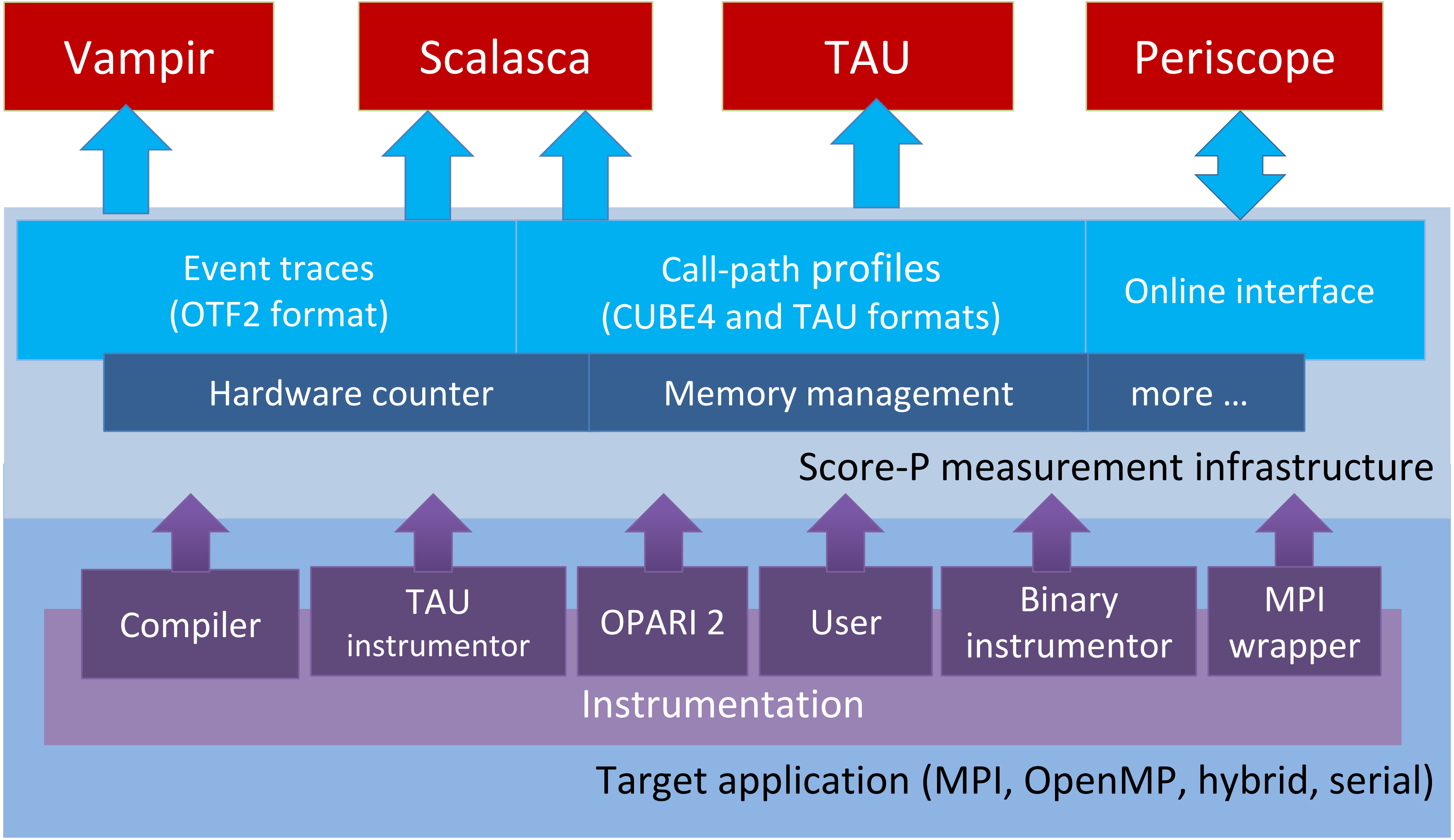
GEFÖRDERT VOM

Bundesministerium
für Bildung
und Forschung

Score-P functionality

- Fundamental tool concepts:
    - Instrumentation (various methods), later sampling
    - Event trace recording
    - Profile generation
    - Online access to profiling data and execution control
- Parallelization methods:
    - MPI
    - OpenMP 3.0
    - Hybrid parallelism (and serial)
- More functionality in the future (Cuda, OmpSs, HMPP, Pthreads, …)
- Analysis tools kept separate on top of Score-P components

Score-P architecture



Vampir

Scalasca

TAU

Periscope

Event traces
(OTF2 format)

Call-path profiles
(CUBE4 and TAU formats)

Online interface

Hardware counter

Memory management

more ...

Score-P measurement infrastructure

Compiler

TAU
instrumentor

OPARI 2

User

Binary
instrumentor

MPI
wrapper

Instrumentation

Target application (MPI, OpenMP, hybrid, serial)

Score-P availability

- Current release is version 1.0.2
  - New BSD license
  - The task profiling features of this presentation will be in the Score-P 1.1 release

- Download: http://www.score-p.org

Goals of the tasking support

- Analysis of task related performance issues
    - Task granularity
    - Task dependency analysis (under development in Scalasca)
- Reconcile existing techniques with tasking
    - No continuous instruction stream per thread anymore
    - Additional level of parallelism and code structure need to be represented
- Generic event model, used for multiple tasking systems
    - Currently, implementation for OpenMP tied tasks
    - OmpSs and HMPP support under development

Task related performance issues

Task granularity

- Tasking offers automated load balancing
  - But introduces task management overhead
- Tasks may be too small
  - The management overhead may cause performance loss
  - Task creation may become a bottleneck
  - Only a fraction of tasks may be too small

    Especially when using recursive task creation structures
  - Identify problematic tasks

    For recursive tasks: Where is the best cut-off?
- Tasks may be too large and too few
  - Reduction of the load balancing effects
  - Similar effects may happen with few, long dependency chains

What data shall we measure?

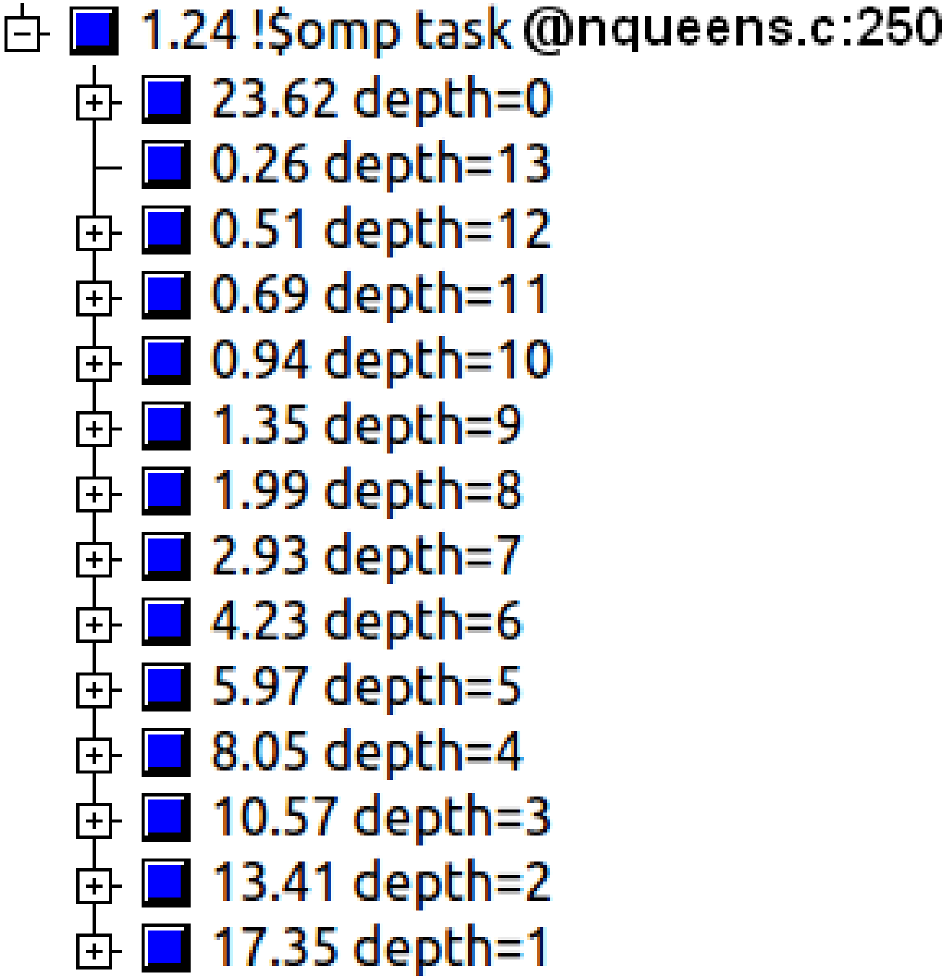
- We want to measure
  - runtime of tasks
  - task creation time and management overhead
  - Number of tasks
- Only a fraction of tasks may have performance issues
  - In the total sum, the effects might be leveled by other tasks
  - Additional statistical information (min,max,median,mean) might help recognizing an issue

How to identify problematic tasks

Provide possibilities to group tasks

- by constructs
- depending on certain parameters (e.g. recursion depth)

mean execution time in µs



- 1.24 !$omp task @nqueens.c:250
  - 23.62 depth=0
  - 0.26 depth=13
  - 0.51 depth=12
  - 0.69 depth=11
  - 0.94 depth=10
  - 1.35 depth=9
  - 1.99 depth=8
  - 2.93 depth=7
  - 4.23 depth=6
  - 5.97 depth=5
  - 8.05 depth=4
  - 10.57 depth=3
  - 13.41 depth=2
  - 17.35 depth=1

Reconcile tasking with existing techniques

Reconcile tasking with existing techniques

No continuous instruction stream per thread anymore

- Distinguish the event stream of each task
- Need to identify task instances
- Track task switches
- For OpenMP tied tasks, we can insert necessary instrumentation

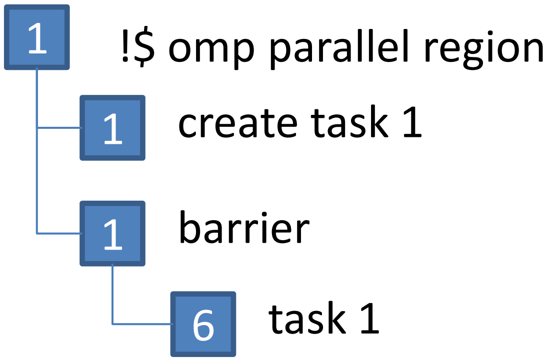
Task data representation

- Additional level of parallelism and code structure
- For Scalasca/Score-P we want to integrate tasks into Cube call trees
- Where shall we place tasks in the call tree?

Display tasks in a Cube4 profile (1)

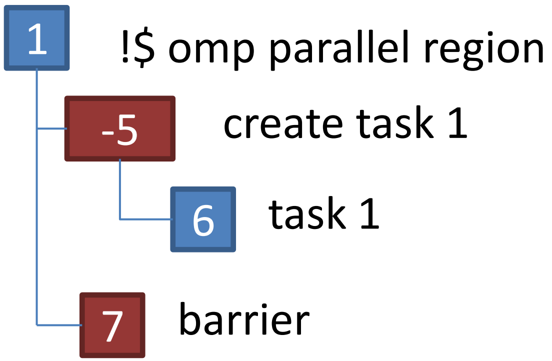
- Require that the inclusive time is the subtree’s sum of exclusive times
- Tasks must appear at execution point in the tree of the implicit task
  - Correct metric attribution
  - Other position may lead to
    - *Negative times for exclusive execution time (and other metrics)*
    - *Appearance of false idle times at synchronization points*

At execution point



1 !$ omp parallel region
1 create task 1
1 barrier
6 task 1

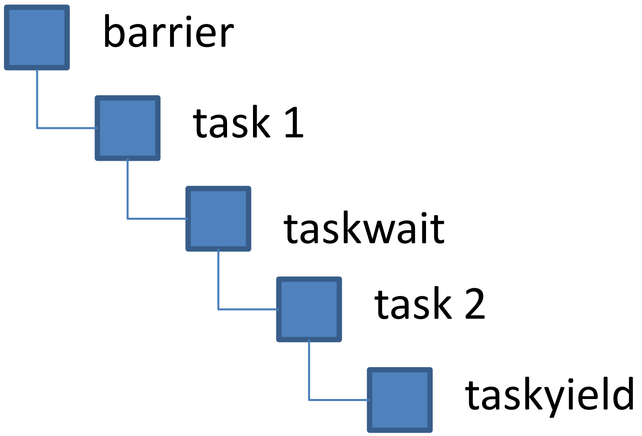
At creation point



1 !$ omp parallel region
-5 create task 1
6 task 1
7 barrier

Display tasks in a Cube4 profile (2)

- All tasks appear as children of the implicit task
- If tasks appear as children in other explicit tasks:
  - *Random execution order leads to incomparable call-tree structure*
  - *Call-tree may become extremely deep*
  - *You might end up with separate node for every task instance*
  - *Could lead to inconsistent call tree*



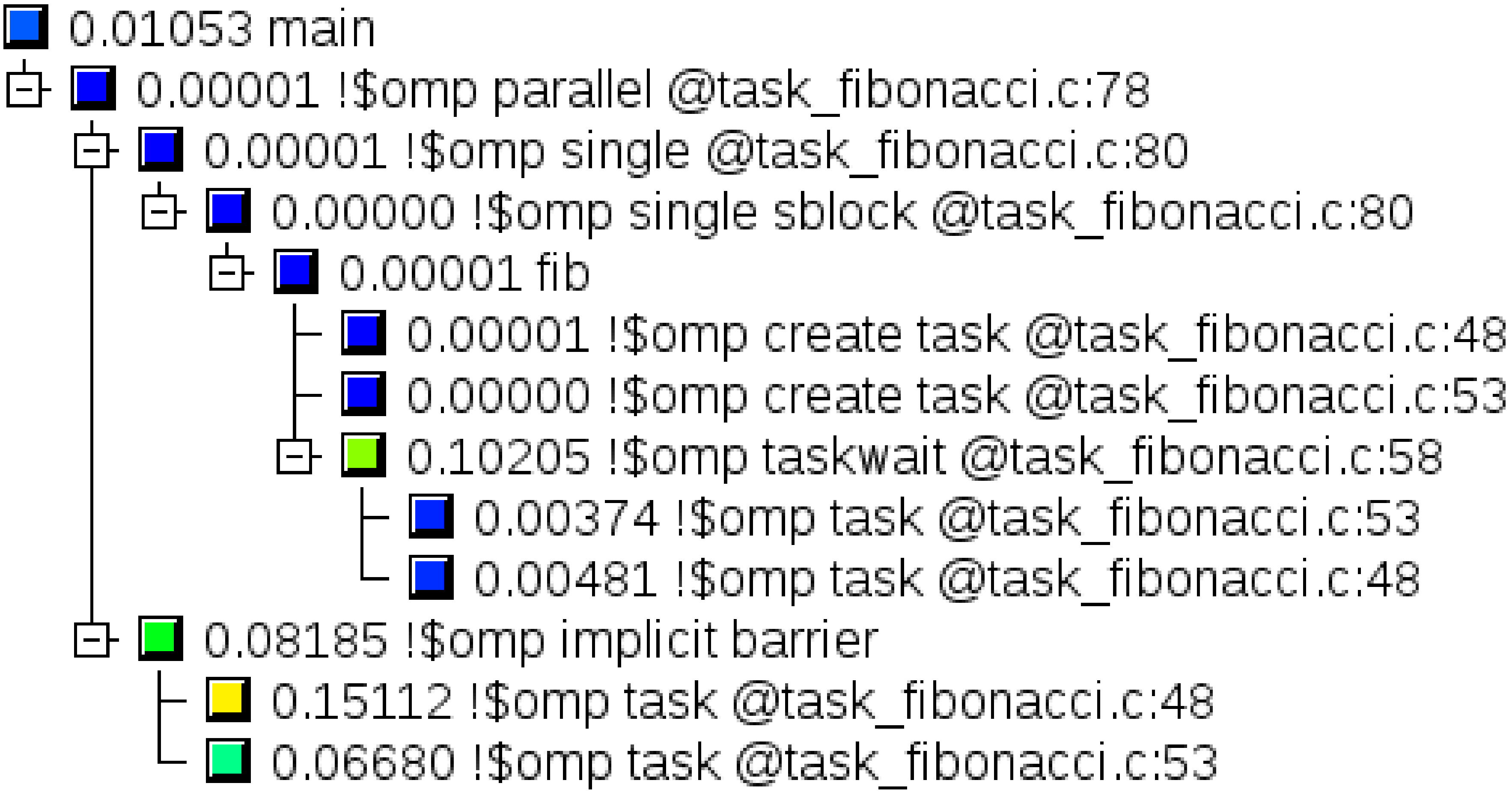
Enter barrier
Start task 1 => enter task 1
Enter taskwait
Start task 2 => enter task2
Enter taskyield
Resume Task 1 => exit taskwait

Display tasks in a Cube4 profile (3)

- A task may be suspended and resumed at another scheduling point
  - How do we count undividable metrics, e.g. visits?

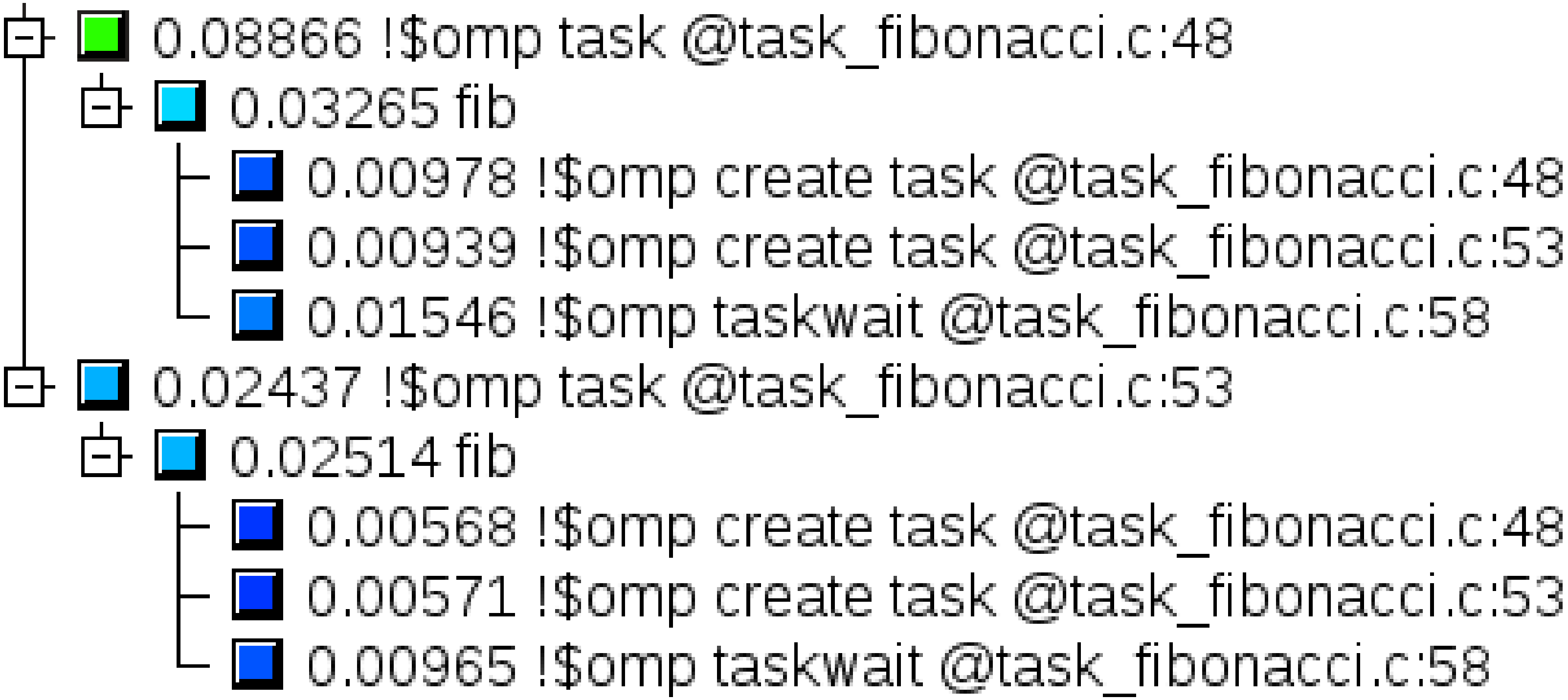
    Similar problem for min, max, sum of squares
  - First event of the resumed task is an exit event
  - We would need to copy the whole call stack of the task
- Solution
  - Leave stub node for task execution at execution point
  - Put task’s inner structure in a separate tree beside the implicit task

Call-tree example (main)



```
🟦 0.01053 main
  🟦 0.00001 !$omp parallel @task_fibonacci.c:78
    🟦 0.00001 !$omp single @task_fibonacci.c:80
      🟦 0.00000 !$omp single sblock @task_fibonacci.c:80
        🟦 0.00001 fib
          🟦 0.00001 !$omp create task @task_fibonacci.c:48
          🟦 0.00000 !$omp create task @task_fibonacci.c:53
          🟩 0.10205 !$omp taskwait @task_fibonacci.c:58
            🟦 0.00374 !$omp task @task_fibonacci.c:53
            🟦 0.00481 !$omp task @task_fibonacci.c:48
    🟩 0.08185 !$omp implicit barrier
      🟨 0.15112 !$omp task @task_fibonacci.c:48
      🟩 0.06680 !$omp task @task_fibonacci.c:53
```

Call-tree example (tasks)



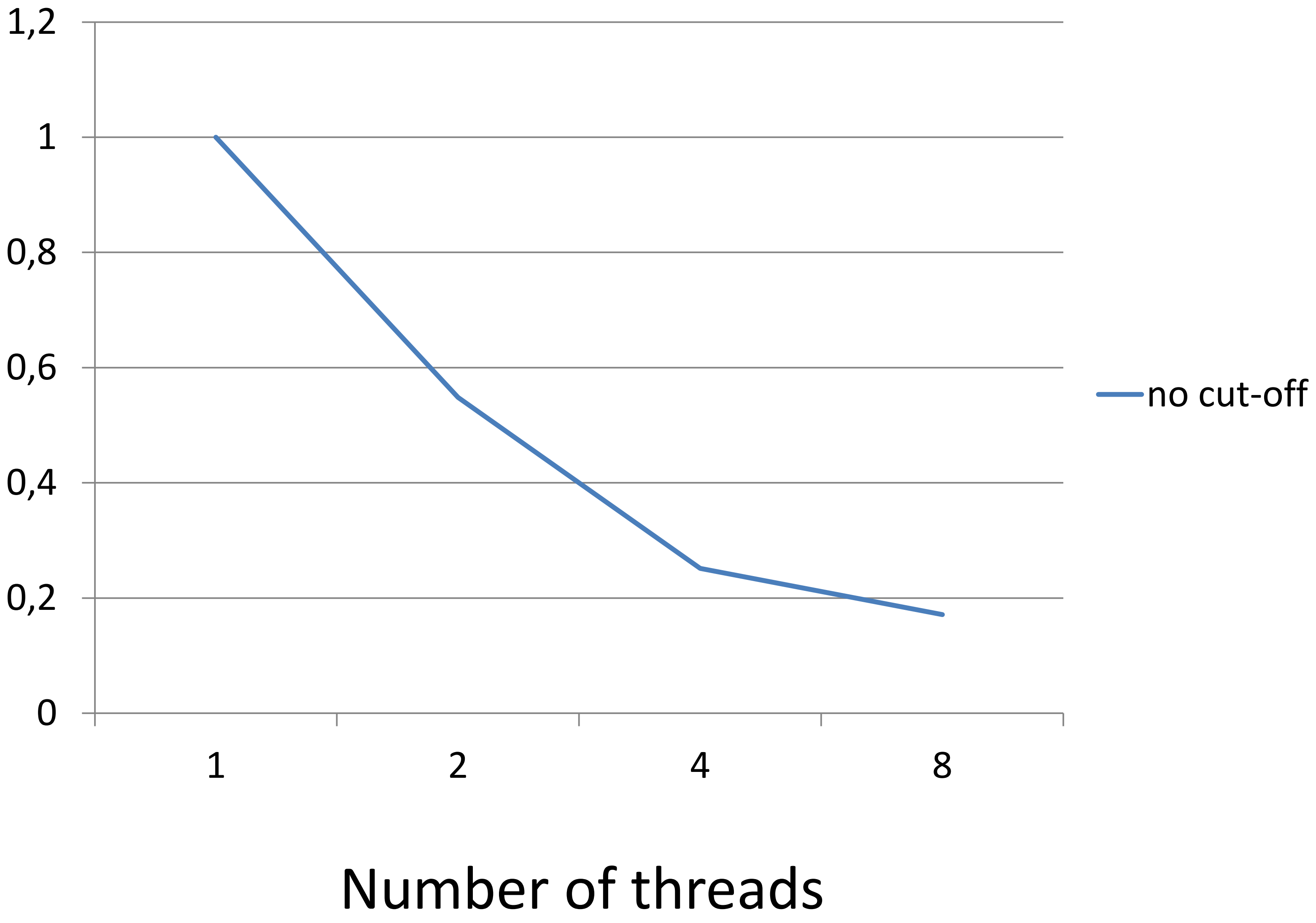
```
0.08866 !$omp task @task_fibonacci.c:48
  0.03265 fib
    0.00978 !$omp create task @task_fibonacci.c:48
    0.00939 !$omp create task @task_fibonacci.c:53
    0.01546 !$omp taskwait @task_fibonacci.c:58
0.02437 !$omp task @task_fibonacci.c:53
  0.02514 fib
    0.00568 !$omp create task @task_fibonacci.c:48
    0.00571 !$omp create task @task_fibonacci.c:53
    0.00965 !$omp taskwait @task_fibonacci.c:58
```

Analysis example

nqueens

- Code of the Barcelona OpenMP Tasking Suite (BOTS)
- Calculate the possibilities to place n queens on an nxn chess board
- BOTS provide multiple versions of the code
  - Analyze the version without cut-off
  - There is also an optimized version with a cut-off
- Runs performed on Juropa using a GNU compiler

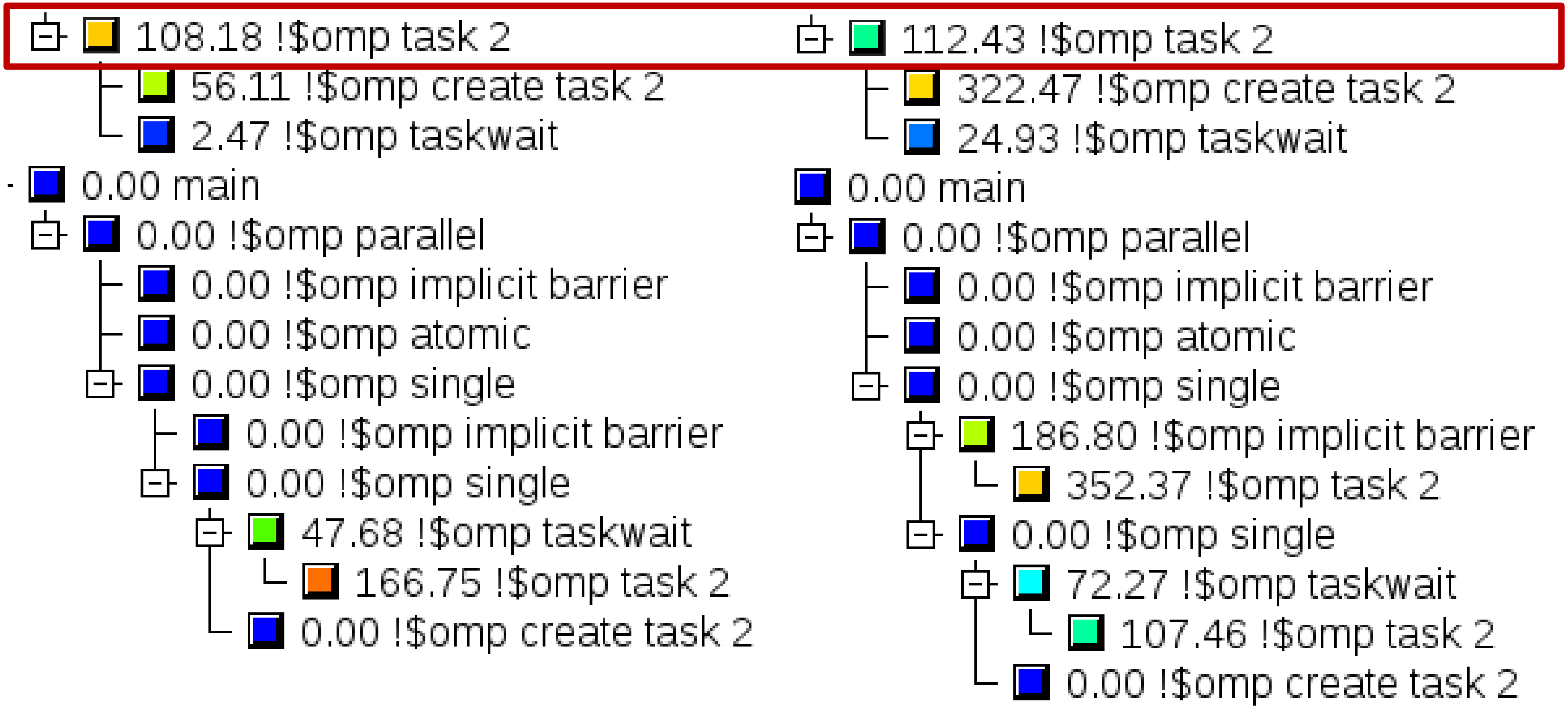
Speedup of nqueens without cut-off (s)



Profile comparison (execution time)

Profile of a run with one thread

Profile of a run with four threads



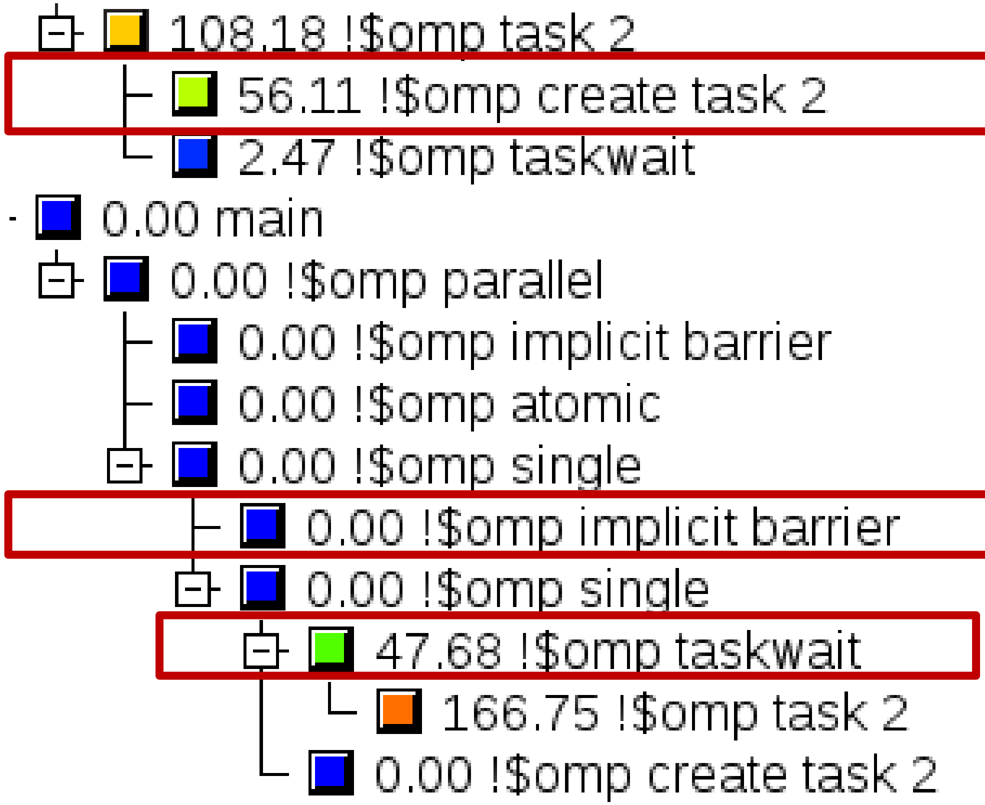
```
108.18 !$omp task 2
  ├ 56.11 !$omp create task 2
  └ 2.47 !$omp taskwait
0.00 main
  0.00 !$omp parallel
    ├ 0.00 !$omp implicit barrier
    ├ 0.00 !$omp atomic
    0.00 !$omp single
      ├ 0.00 !$omp implicit barrier
      0.00 !$omp single
        47.68 !$omp taskwait
          └ 166.75 !$omp task 2
        └ 0.00 !$omp create task 2
```

```
112.43 !$omp task 2
  ├ 322.47 !$omp create task 2
  └ 24.93 !$omp taskwait
0.00 main
  0.00 !$omp parallel
    ├ 0.00 !$omp implicit barrier
    ├ 0.00 !$omp atomic
    0.00 !$omp single
      186.80 !$omp implicit barrier
        └ 352.37 !$omp task 2
      0.00 !$omp single
        72.27 !$omp taskwait
          └ 107.46 !$omp task 2
        └ 0.00 !$omp create task 2
```

Sum of execution time of user code over all threads stays nearly the same

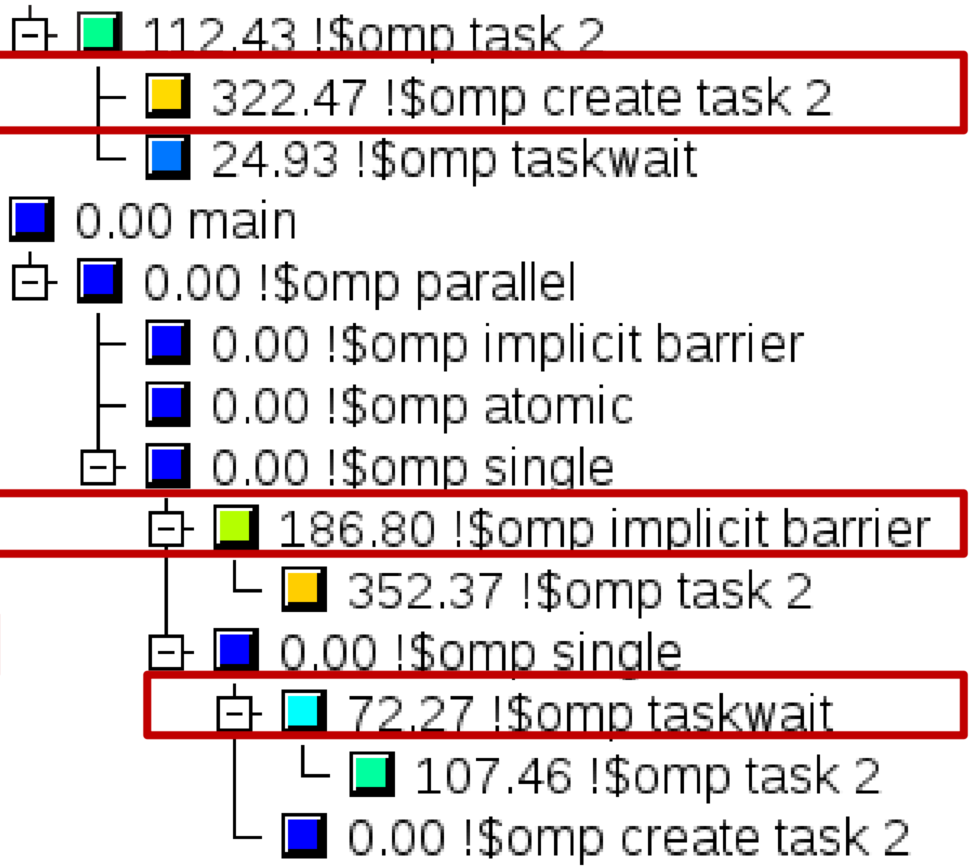
Profile comparison (execution time)

Profile of a run with one thread



```
⊟ ■ 108.18 !$omp task 2
    ├ ■ 56.11 !$omp create task 2
    └ ■ 2.47 !$omp taskwait
· ■ 0.00 main
  ⊟ ■ 0.00 !$omp parallel
      ├ ■ 0.00 !$omp implicit barrier
      ├ ■ 0.00 !$omp atomic
      ⊟ ■ 0.00 !$omp single
          ├ ■ 0.00 !$omp implicit barrier
          ⊟ ■ 0.00 !$omp single
              ⊟ ■ 47.68 !$omp taskwait
              │   └ ■ 166.75 !$omp task 2
              └ ■ 0.00 !$omp create task 2
```

Profile of a run with four threads



```
⊟ ■ 112.43 !$omp task 2
    ├ ■ 322.47 !$omp create task 2
    └ ■ 24.93 !$omp taskwait
■ 0.00 main
⊟ ■ 0.00 !$omp parallel
    ├ ■ 0.00 !$omp implicit barrier
    ├ ■ 0.00 !$omp atomic
    ⊟ ■ 0.00 !$omp single
        ⊟ ■ 186.80 !$omp implicit barrier
        │   └ ■ 352.37 !$omp task 2
        ⊟ ■ 0.00 !$omp single
            ⊟ ■ 72.27 !$omp taskwait
            │   └ ■ 107.46 !$omp task 2
            └ ■ 0.00 !$omp create task 2
```

Additional time due to management overhead

Tasks by recursion level

Depth level	Mean time	Sum	Number of tasks
0	23.6 µs	0.0003 s	14
1	17.4 µs	0.0034 s	196
2	13.4 µs	0.0293 s	~ 2,000
3	10.6 µs	0.2019 s	~19,000
4	8.05 µs	1.086 s	~135,000
5	5.97 µs	4.520 s	~750,000
6	4.23 µs	14.31 s	~3,400,000
7	2.93 µs	34.25 s	~11,700,000
8	1.98 µs	61.56 s	~31,000,000
9	1.35 µs	83.01 s	~61,000,000
10	0.94 µs	83.48 s	~89,000,000
11	0.69 µs	62.42 s	~91,000,000
12	0.51 µs	32.26 s	~63,000,000
13	0.26 µs	7.145 s	~27,000,000

Tasks by recursion level

Mean task creation time approx. 0.85 µs

	Depth level	Mean time	Sum	Number of tasks
	0	23.6 µs	0.0003 s	14
	1	17.4 µs	0.0034 s	196
>5 % Overhead	2	13.4 µs	0.0293 s	~ 2,000
	3	10.6 µs	0.2019 s	~19,000
>10 % Overhead	4	8.05 µs	1.086 s	~135,000
	5	5.97 µs	4.520 s	~750,000
>20 % Overhead	6	4.23 µs	14.31 s	~3,400,000
	7	2.93 µs	34.25 s	~11,700,000
	8	1.98 µs	61.56 s	~31,000,000
	9	1.35 µs	83.01 s	~61,000,000
	10	0.94 µs	83.48 s	~89,000,000
>100 % Overhead	11	0.69 µs	62.42 s	~91,000,000
	12	0.51 µs	32.26 s	~63,000,000
	13	0.26 µs	7.145 s	~27,000,000

JÜLICH
FORSCHUNGSZENTRUM

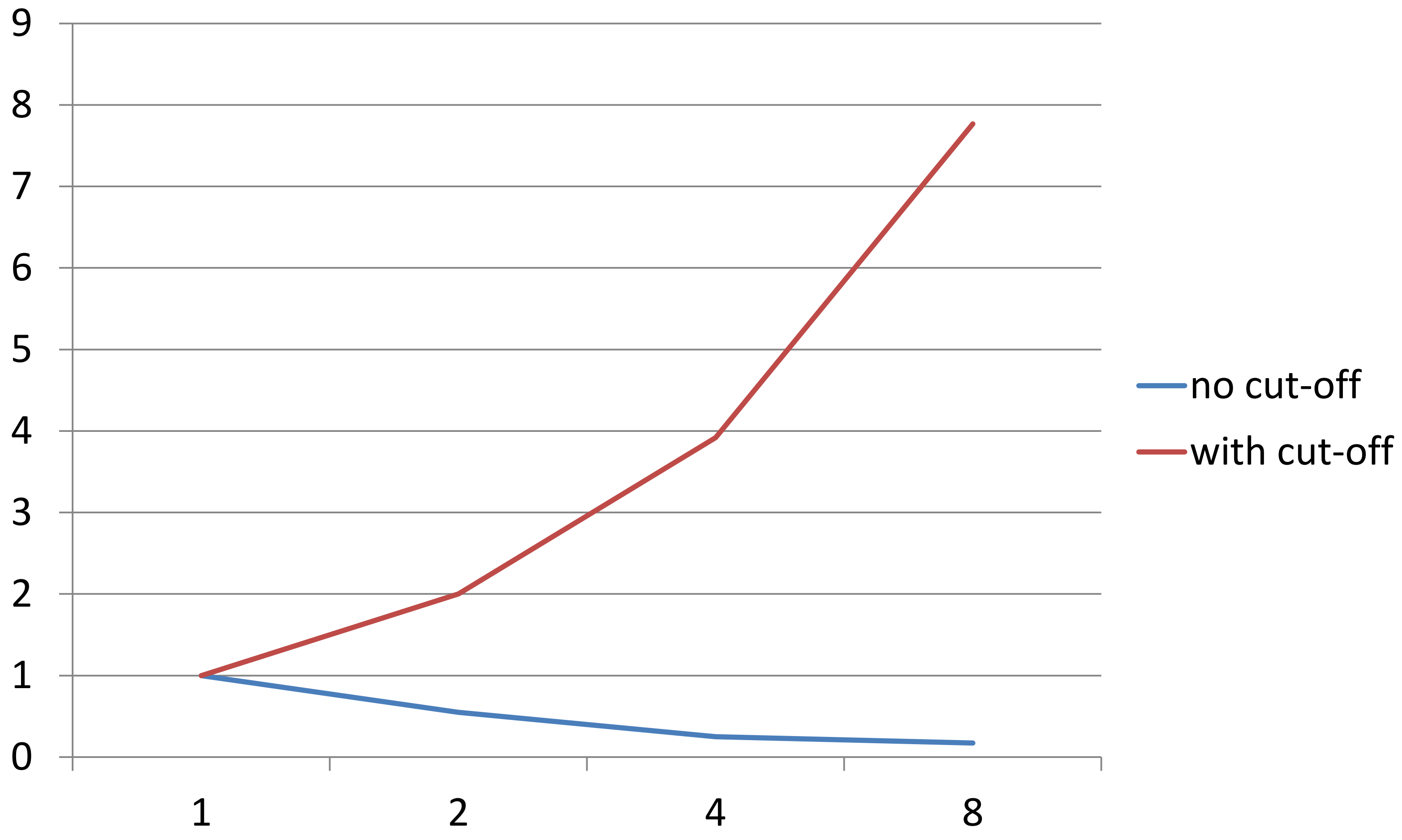
Tasks by recursion level

Mean task creation time approx. 0.85 µs

Depth level	Mean time	Sum	Number of tasks
0	23.6 µs	0.0003 s	14
1	17.4 µs	0.0034 s	196
>5 % Overhead 2	13.4 µs	0.0293 s	~ 2,000
3	10.6 µs	0.2019 s	~19,000
>10 % Overhead 4	8.05 µs	1.086 s	~135,000
5	5.97 µs	4.520 s	~750,000
6	4.23 µs	14.31 s	~3,400,000

- Let us target less than 5% management overhead per task
- 210 tasks may be too little for proper load balancing
- Upper levels do not contribute significant amount of execution time
- Tasks in last level will grow due to merge with children
- Compromise: Cut-off at level 3

Resulting speedup



Number of threads

Future Work

- Currently, only OpenMP tied tasks are supported
  - Ongoing work on HMPP and OmpSs support
  - Hopefully, a new OpenMP tools interface provides necessary information to support untied tasks, too

- Trace analysis of tasks with Scalasca
  - Extend for additional patterns
  - Task dependency analysis