How to reconcile event-based performance analysis with tasking in OpenMP

Daniel Lorenz

Forschungszentrum Jülich

IWOMP 2010

Outline

Introduction

Problem analysis

Tied tasks

- Task identifiers
- Creation hierarchy

Approaches for untied tasks

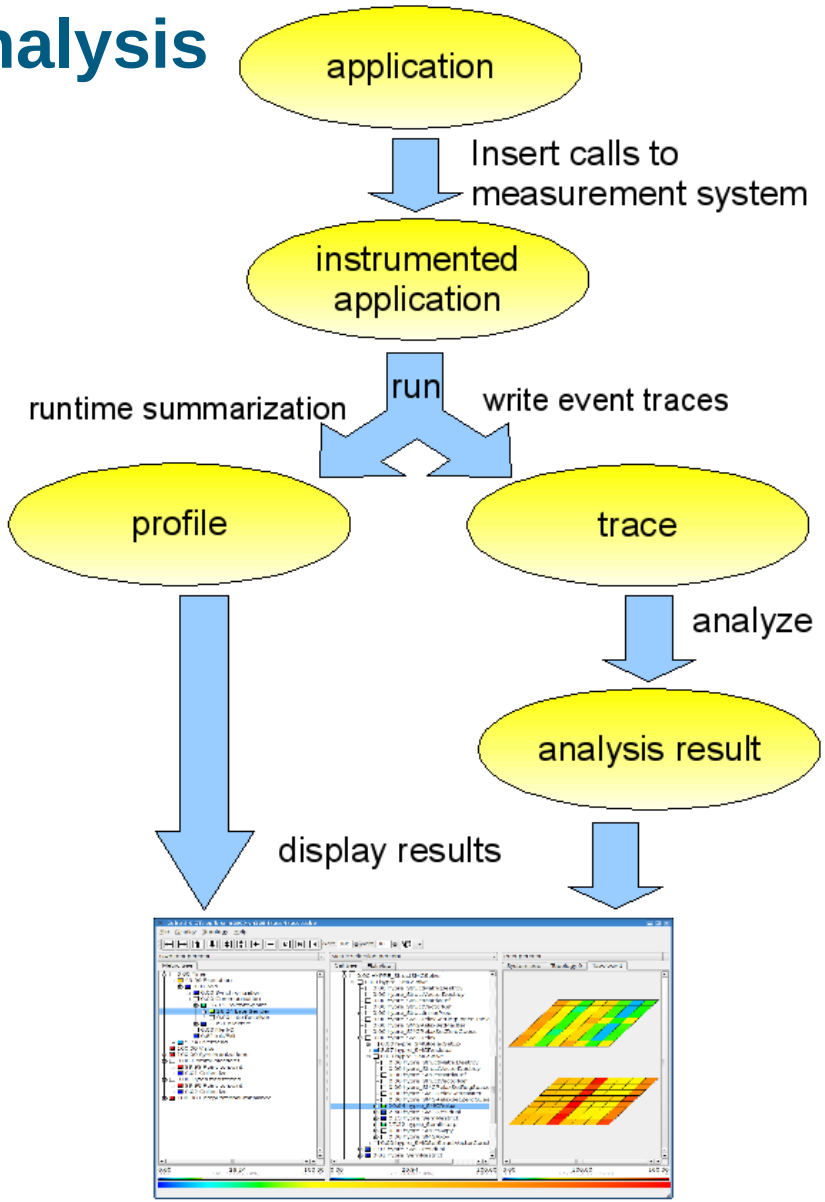
Integration in OPARI

Evaluation

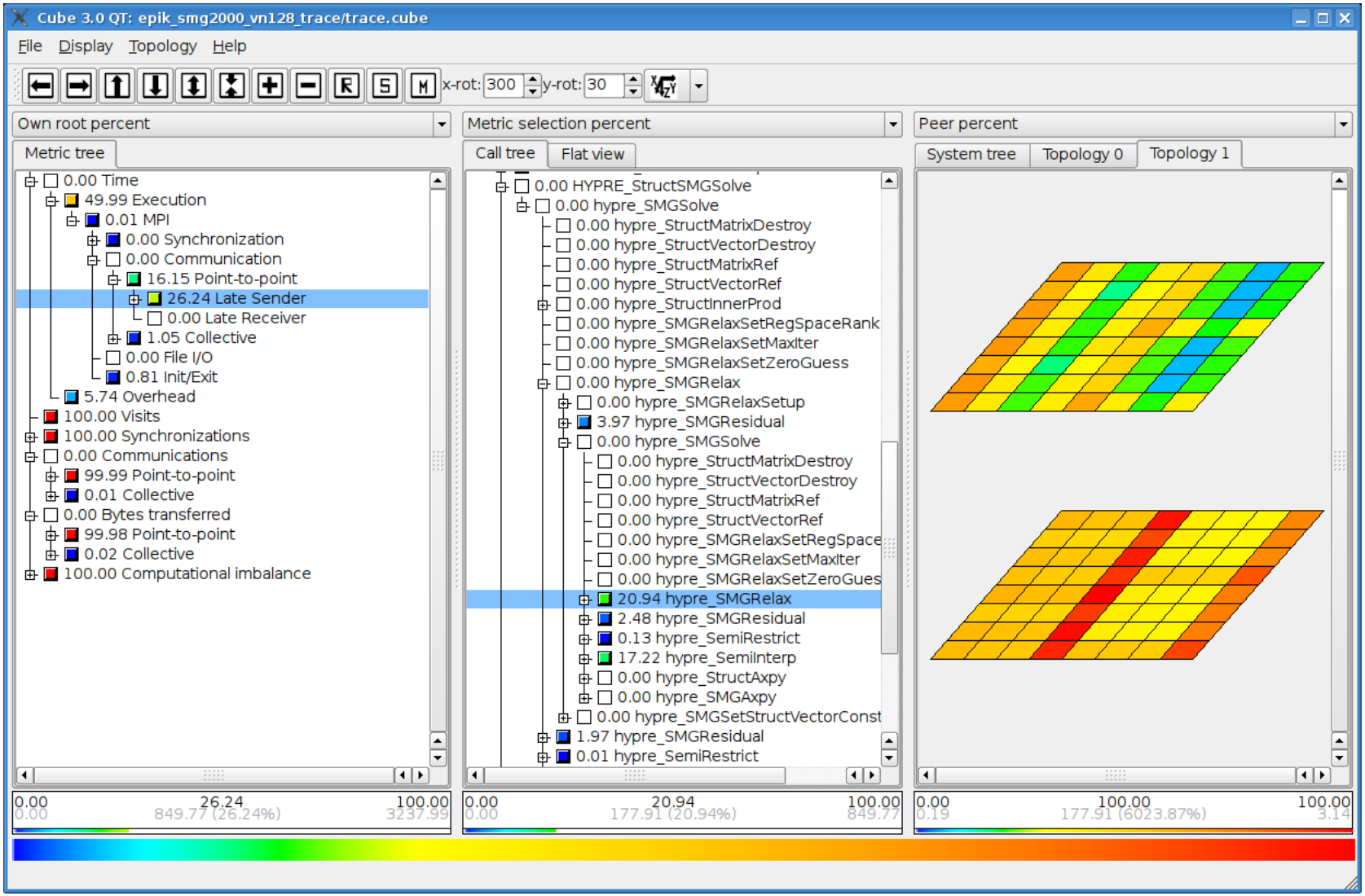
Conclusion and future work

Event-based performance analysis

- Instrument e.g. enter and exit points of functions
- Record performance data on events
- Construct a call tree from enter and exit calls
- Analysis relies on correct nesting of enter and exit events



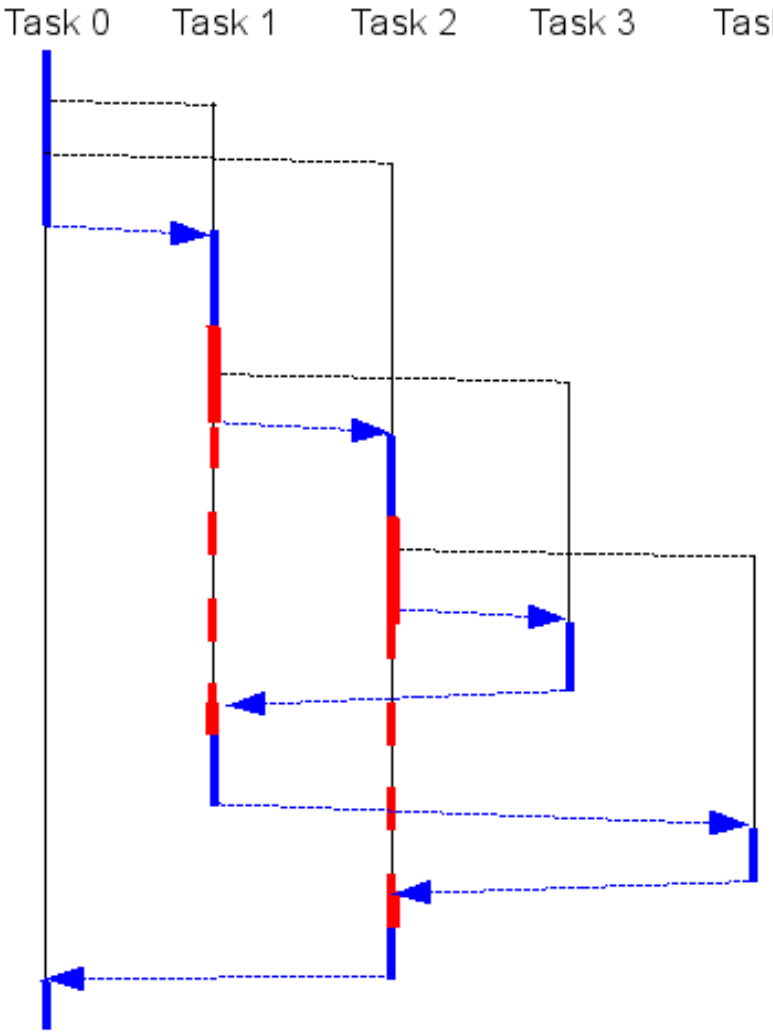
Cube display



OpenMP tasks

- Introduced in OpenMP 3.0
- Concurrent execution
- Two types:
  - Tied tasks
    - *Suspended only in scheduling points*
    - *Are always resumed by the same thread that executed the task before*
  - Untied tasks
    - *Can be interrupted anywhere*
    - *Can be resumed by another thread*

Incorrect nesting of events



Create and suspend Task 1.
Create and suspend Task 2.
Encounter taskwait statement.
Execute enter event for f1().
Create and suspend Task 3.
Encounter taskwait statement.
Execute enter event for f2().
Create and suspend Task 4.
Encounter taskwait statement.
Task 3 is done.
Execute exit event for f1().
Task 1 is done.
Task 4 is done.
Execute exit event for f2().
Task 2 is done.

How can the nesting problem be solved?

- Identify task instances
  - Separate event sequences for each task
- Create events on task suspension and resumption
  - Allows to identify suspended intervals
  - Requires task instance identification

Identify task instances for tied tasks

Idea:

- Emulate task local storage
  - Unchanged on function calls and task suspension/resumption
  - Different for each task instance

- Store an unique identifier in a task local storage
  - Generate identifiers from thread identifier and threadprivate counter

Task local storage for tied tasks

- Tied tasks can be suspended/resumed only at scheduling points
  - Use thread local storage to store data between scheduling points
  - Save value to local variable before scheduling point
  - Restore value from local to thread local storage after scheduling point

  - Measurement library can obtain task identifier from thread local storage on every event

Task creation hierarchy

- Obtain information which task instance is the creator of a given task
  - Useful for, e.g., analysis of wait dependencies

- Idea:
  - Pass the identifier of parent as firstprivate to child
  - Insert event on task creation which passes the parent identifier to measurement system

Instrumentation

At global scope:

```
int64_t current_task_id = ROOT_TASK_ID;
#pragma omp threadprivate(current_task_id)
```

Instrumented task statement (before)

Before instrumentation:

```
#pragma omp task {
  // do something
}
```

Instrumented task statement (after)

After Instrumentation:

```
{
  int64_t old_task_id = current_task_id;
  task_create_begin();
  #pragma omp task firstprivate (old_task_id) {
    current_task_id = get_new_task_id();
    task_begin(old_task_id);
    // do something
    task_end();
  }
  current_task_id = old_task_id;
  task_create_end();
}
```

Untied tasks

- Can be resumed by other thread
  - Thread identifier in local storage still valid
- Can be suspended at any point
  - Can not save the task identifier from thread local storage to local variable
  - Can not emulate task local storage

Approaches for untied tasks

- Could make all tasks tied by instrumentation
  - Allows correct tracking
  - May change behavior of application
- Some compilers (e.g. Sun) do suspend untied tasks only on scheduling points
  - Task local storage emulation works
- Extend OpenMP specification

Proposed extensions to OpenMP

- Provide task identifiers by OpenMP implementation
  - Implementation has internal task identifier anyway
- Possibility to register a callback function which is called on task resumption

Automated instrumentation with OPARI

- OPARI:
  - Source code instrumenter
  - Instruments OpenMP constructs

- Extended OPARI:
  - Instrument task and taskwait constructs
  - Added task identifier handling to scheduling points (parallel regions, barriers)
  - Implemented for instrumenting C/C++ codes

Runtime overhead of instrumentation

- Two tests
  - Artificial program
  - Flexible Image Retrieval Engine (FIRE) code

- Instrumentation via extended OPARI
- Maintained task identifiers
- Evaluated instrumentation only (no data recording)

Artificial test program

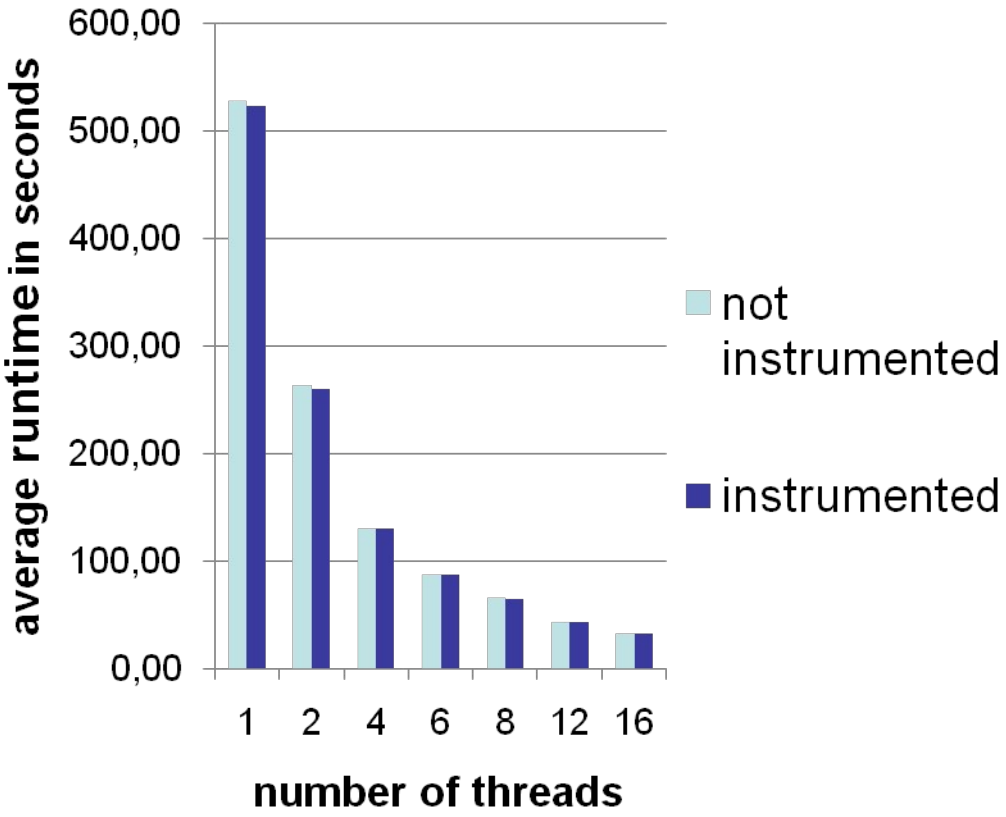
- Created 10,000,000 tasks per thread
- Task only increment one integer
- Run with 4 threads

original program	instrumented program	absolute overhead	relative overhead
1.89 s	2.50 s	0.61 s	32.3%

- Absolute overhead negligible
- Relative overhead to task creation 32.3 %

FIRE code

- Flexible Image Retrieval Engine (FIRE)
- RWTH Aachen University
- 35,000 lines C++
- Finds similar images in a database
- 18 query images
- 1000 images in database
- 18,000 tasks
- IBM eServer LS42
- AMD Opteron 8356 processors



FIRE code results

	runtime (sec.)		speedup			
#threads	not instrumented	instrumented	speedup not instrumented	speedup instrumented	overhead in %	overhead in sec.
1	522,57	527,56	1,00	1,00	0,96%	4,99
2	259,55	262,64	2,01	2,01	1,19%	3,09
4	129,52	129,93	4,06	4,03	0,32%	0,41
6	86,42	86,43	6,10	6,05	0,01%	0,01
8	64,86	65,13	8,10	8,06	0,41%	0,27
12	43,13	43,00	12,27	12,12	-0,30%	0,13
16	32,12	32,43	16,27	16,27	0,95%	0,31

Conclusion

- Performance analysis of applications with tasks require:
  - Identification of task instances
  - Notification of task suspend/resume
- For tied tasks:
  - Identifiers can be stored in emulated task local storage
  - Notifications can be obtained from instrumentation of OpenMP constructs
- For untied tasks:
  - Exploit fact that some compilers reschedule only at special points
  - For general portable solution extension of OpenMP specification required

Future work

- Implement presented approach for Fortran
- Develop measurement system to record task data
- Create analysis patterns for tasks