Task Parallelism on the SCC

Andreas Prell
andreas.prell@uni-bayreuth.de

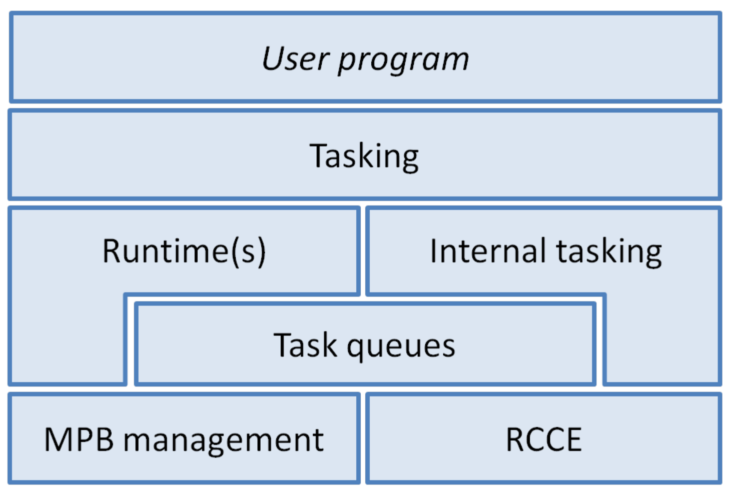
Thomas Rauber
thomas.rauber@uni-bayreuth.de

Motivation

Task parallel programming has become a popular and effective programming model for multicores

- High-level task abstraction (threads are implementation detail)
- All potential parallelism is expressed in terms of tasks
- Runtime system takes care of assigning tasks to threads

What about task parallel programming on the SCC?
We have implemented a tasking environment on top of RCCE



Tasking on the SCC

The SCC's on-chip MPB memory allows efficient task movement between cores
- MPB task queues based on one-sided put/get operations
- Small number of test-and-set registers is somewhat restrictive

Runtime system schedules tasks and performs load balancing
- Work-sharing of private tasks using a central MPB queue
- Work-stealing between MPB deques

Task synchronization via *taskbarrier*, *taskwait* [1], and *futures*
- *taskbarrier:* waits for the completion of all pending tasks
- *taskwait*: waits for the completion of all immediate child tasks
- *future*: task that computes a result, forcing a future means waiting until the result is available

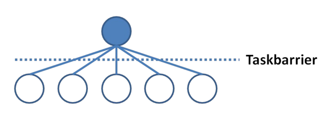
Compiler support desirable → work in progress

[1] E. Ayguadé *et al.* The Design of OpenMP Tasks. In IEEE TPDS, vol. 20, pp. 404-418, 2009

Preliminary Experimental Results

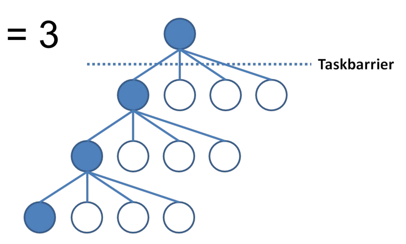
❶ Simple Producer-Consumer (SPC)
A single producer (worker *ID* 0) spawns *n* consumer tasks, which compute for time *t*.

Example: *n* = 5



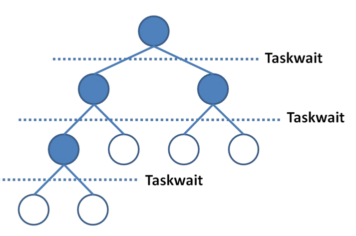
❷ Bouncing Producer-Consumer (BPC) [2]
A variation of the producer-consumer benchmark with two kinds of tasks, producer and consumer tasks. Each producer task creates another producer task followed by *n* consumer tasks, until a depth of *d* is reached. Consumer tasks perform some computation for time *t*.

Example: *n* = 3, *d* = 3

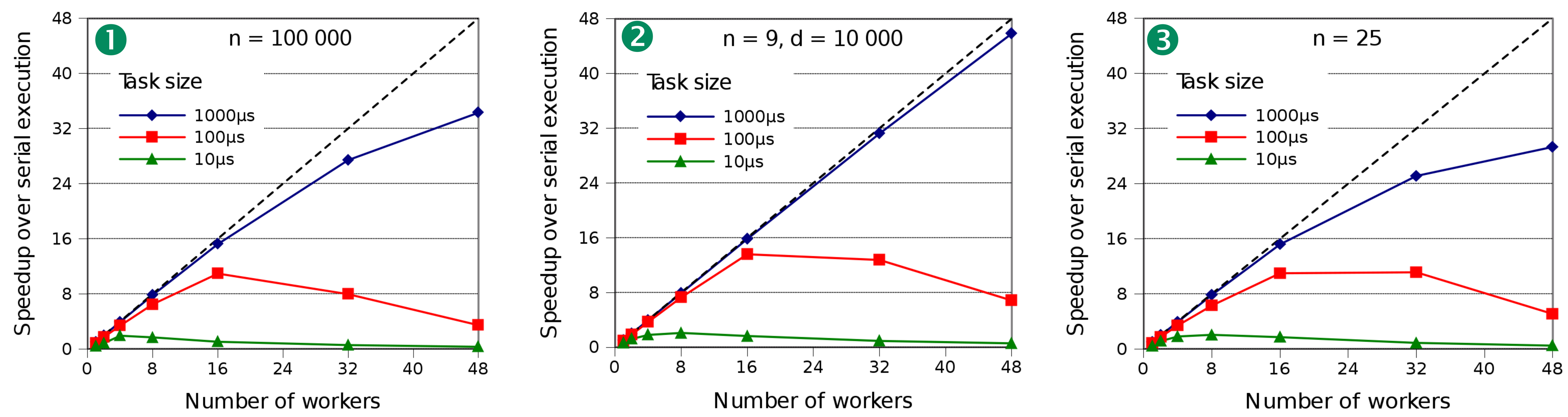


❸ Fibonacci-like tree recursion
Each task *n* ≥ 2 spawns two child tasks *n*-1 and *n*-2 and waits for their completion. Leaf tasks *n* < 2 end the recursion and compute for time *t*.

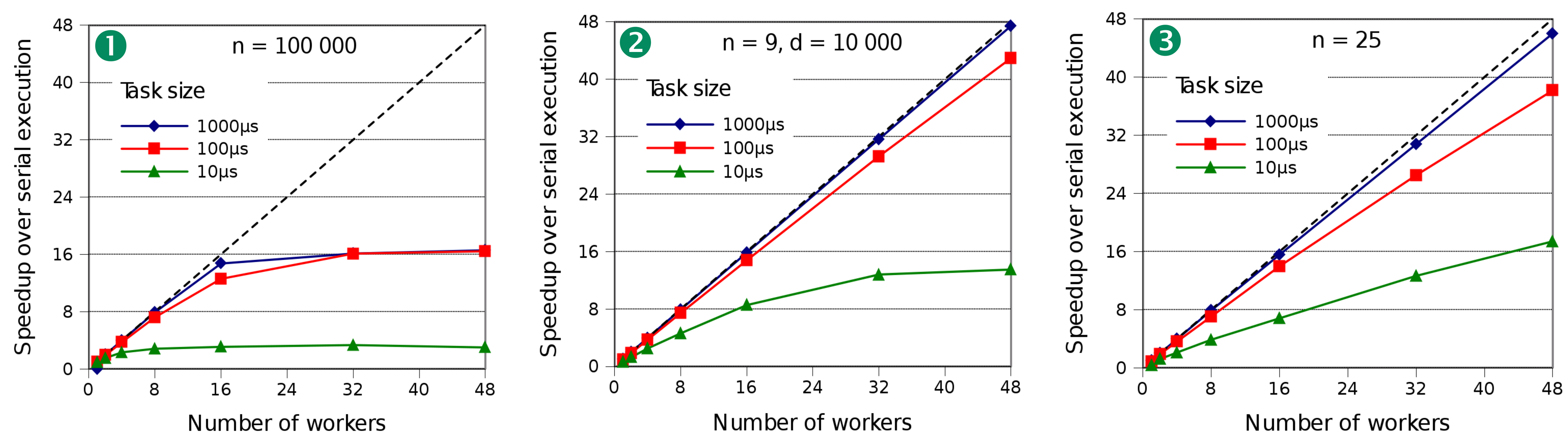
Example: *n* = 4



Work-sharing



Work-stealing



[2] J. Dinan *et al.* Scalable Work Stealing. In SC '09, pp. 53:1-53:11, 2009

Summary of Results

Work-sharing
- Poor choice if parallelism is fine-grained
- Can be practical for certain types of workloads

Work-stealing
- Much better scalability than work-sharing
- Current implementation puts pressure on MPB memory
- Tradeoff between performance and on-chip memory consumption

Outlook

Work-sharing and work-stealing schedulers are a good starting point for further runtime system research

- Message-passing schedulers? [3]
- Shared state ↓ Scalability ↑
- Research challenge: runtime systems should be performance portable to other (future) manycore platforms

[3] D. Sanchez *et al.* Flexible Architectural Support for Fine-Grain Scheduling. In ASPLOS '10, pp. 311-322, 2010

Applied Computer Science *II* – Parallel and Distributed Systems
Department of Computer Science, University of Bayreuth, Germany