Task Parallel Programming Support for the Single-Chip Cloud Computer

Andreas Prell andreas.prell@uni-bayreuth.de, **Thomas Rauber** thomas.rauber@uni-bayreuth.de

UNIVERSITÄT BAYREUTH

Motivation

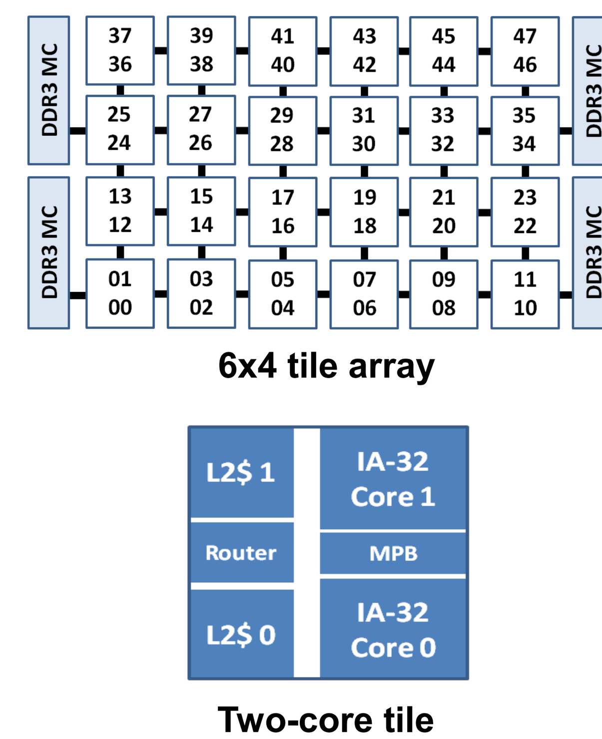
Task parallel programming is a popular and effective programming model for multicores

- 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

Intel Single-Chip Cloud Computer (SCC) [1]

- Manycore software research vehicle
- Tiled architecture, 48 Pentium-class IA-32 cores
- 384 KB shared on-chip SRAM (MPB), private/shared off-chip DRAM
- Native programming model: message passing (think MPI) [2]
- Communication through non-cache-coherent shared memory

What about task parallel programming on the SCC?
Need runtime support for dynamic task parallelism



6x4 tile array

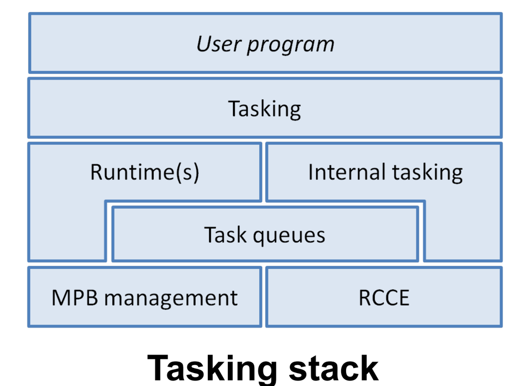
Two-core tile

[1] J. Howard *et al.* A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In ISSCC '10, 2010
[2] T. G. Mattson *et al.* The 48-core SCC Processor: the Programmer's View. In SC '10, 2010

Tasking on the SCC

Shared on-chip memory allows efficient task movement between cores

- Task queue implementation based on one-sided put/get operations
- Small number of test-and-set registers (48) required for mutual exclusion is somewhat restrictive (no atomic operations on the SCC!)



Tasking stack

Runtime system schedules tasks and performs load balancing

- Work-sharing of private tasks using a central task queue
- Work-stealing between local deques

`async compute()` creates a task to run `compute()` *asynchronously* with the calling code

Task synchronization via *taskbarrier*, *taskwait* [3], and *futures*

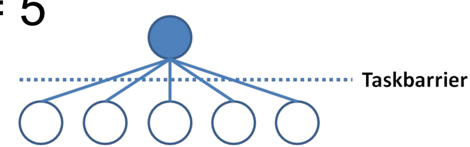
- *taskbarrier:* waits for the completion of all pending tasks
- *taskwait*: waits for the completion of all immediate child tasks
- *future*: task that returns a result, forcing a future means waiting until the result is available

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

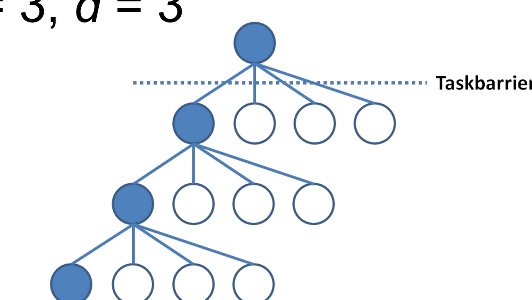
Preliminary Experimental Results

Stress tests ● spawns child tasks ○ computes for time *t*

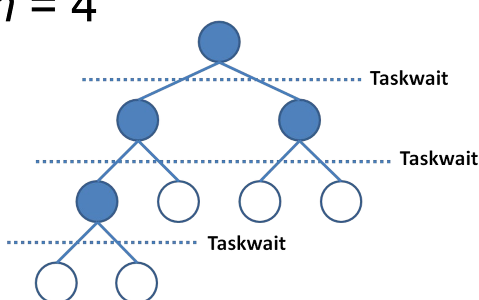
Simple Producer-Consumer (SPC)
Example: $n = 5$



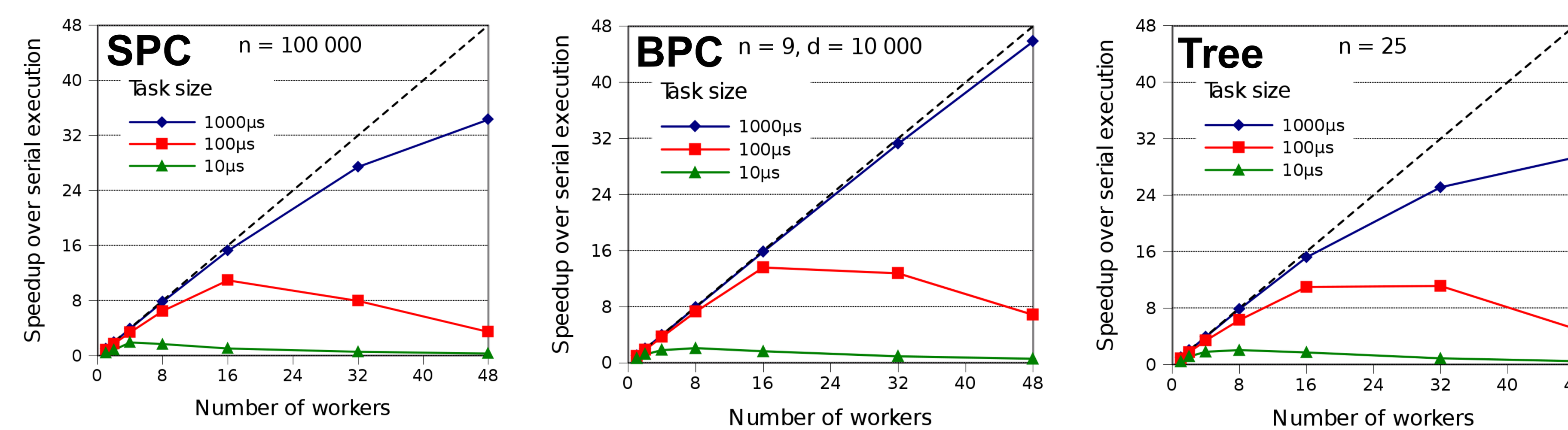
Bouncing Producer-Consumer (BPC) [4]
Example: $n = 3$, $d = 3$



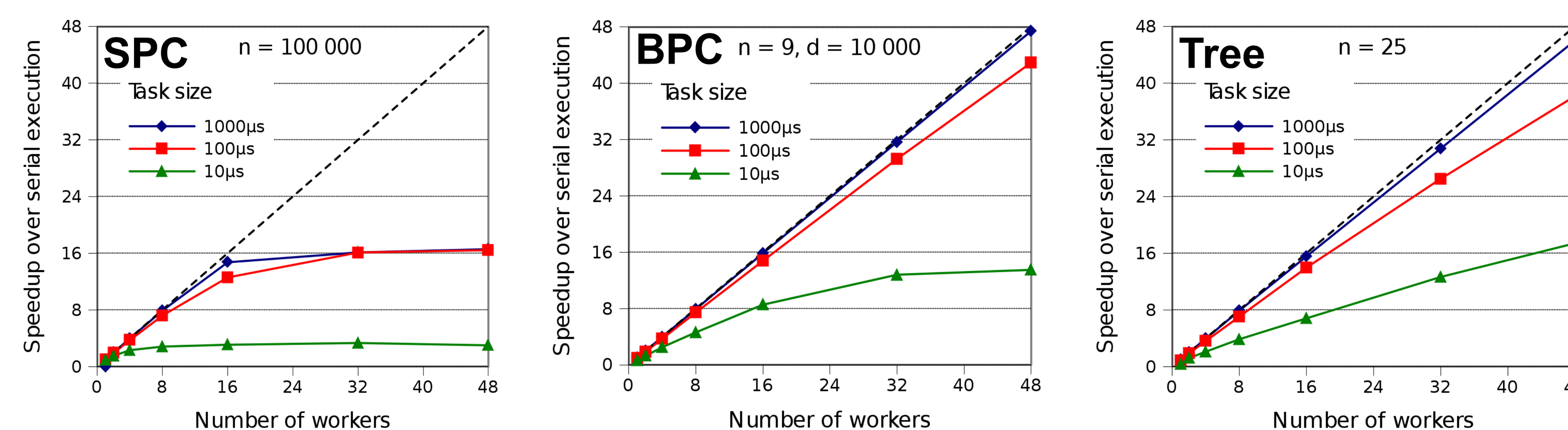
Fibonacci-like tree recursion (Tree)
Example: $n = 4$



Work-sharing



Work-stealing

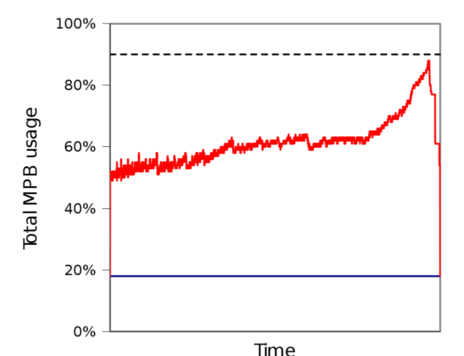


[4] J. Dinan *et al.* Scalable Work Stealing. In SC '09, pp. 53:1-53:11, 2009

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 on-chip memory
- Tradeoff between performance and on-chip memory consumption

Figure: On-chip memory consumption of work-stealing running the tree-recursive benchmark with 48 workers. The task queues, which were configured to have a maximum size of 10, account for roughly 20% of the available memory. To avoid overflows, task execution is serialized if a worker has allocated 90% of its local memory.

Future Work

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

- Message-passing schedulers [5]
- Reduce shared state to improve scalability
- Research challenge: runtime systems should be performance portable to other (future) manycore platforms

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