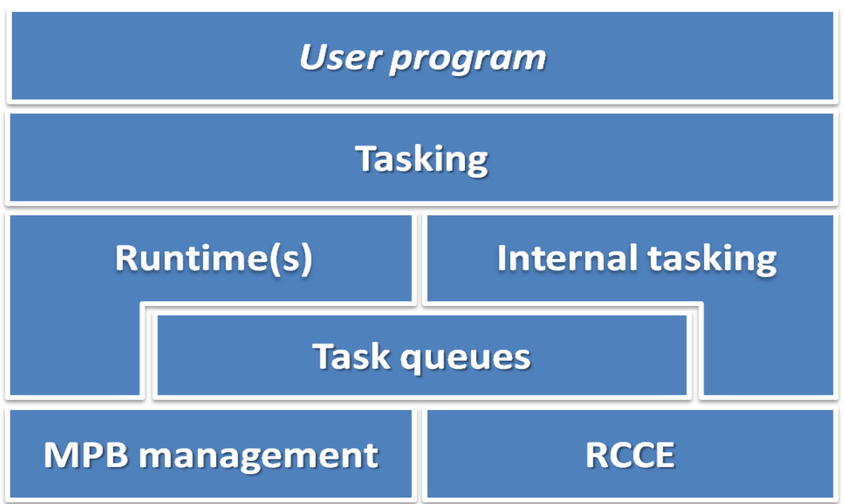
Task Parallelism on the SCC

Andreas Prell and Thomas Rauber
University of Bayreuth, Germany
andreas.prell@uni-bayreuth.de

Porting the task parallel programming model

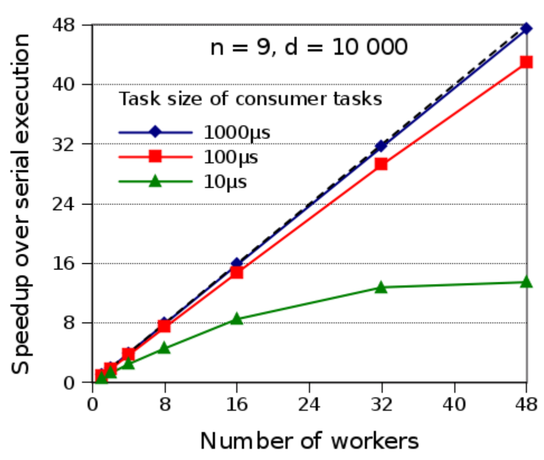
Task parallel programming has become a popular approach to multicore programming. An interesting question is thus: how does this model translate to novel architectures, such as Intel's SCC, and will it be able to efficiently support applications with fine-grained irregular parallelism?
There are many ways to load-balance a computation on a processor that supports both distributed and shared memory spaces. We try to find out which runtime strategies are the most efficient, and which of them can be used in a (distributed) manycore environment. How does work-stealing look like on the SCC and can it provide scalable performance?

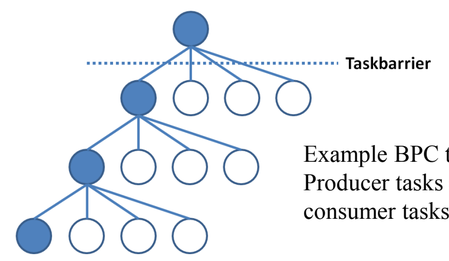
A Tasking Runtime System for the SCC



The runtime is the central component that implements the scheduler. We have currently two different implementations: a simple *work-sharing* scheduler based on private and shared off-chip memory and a *work-stealing* scheduler that makes heavy use of the on-chip Message Passing Buffers (MPBs). The results on the right were obtained with the work-stealing runtime.
Much of our future work will focus on different load balancing strategies that can be adapted to work in (possibly distributed) manycore environments. To hide the complexity of the tasking interface, we are thinking about introducing ASYNC/WAIT language constructs to denote parallelism and synchronization between tasks. A source-to-source translator could then be used to transform the code and insert appropriate calls to the runtime library.

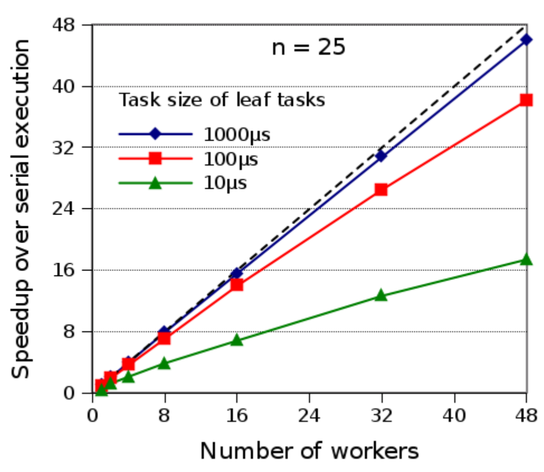
Preliminary Experimental Results

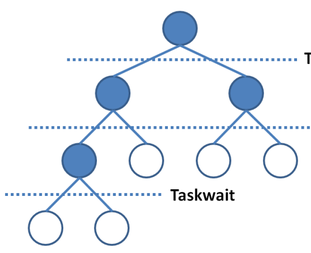




Example BPC task tree for $n = 3$, $d = 3$
Producer tasks are drawn as filled circles, consumer tasks as empty circles

Bouncing Producer-Consumer (BPC) benchmark [1]
There are two kinds of tasks: *producer tasks* and *consumer tasks*. Each producer task creates another producer task followed by n consumer tasks, up to a depth of d. This benchmark stresses the ability of the runtime scheduler to find work and perform load balancing.





Example Fib-like task tree for $n = 4$
Non-leaf tasks are drawn as filled circles, leaf tasks as empty circles

Fibonacci-like tree-recursive benchmark
Each task $n \geq 2$ creates two child tasks n-1 and n-2 and waits for their completion. Leaf tasks $n < 2$ end the recursion by performing some computation. This benchmark includes the runtime overhead that is incurred to keep track of parent-child dependencies.

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