Middleware for Many-Cores
Why it is needed and what functionality it should provide

Randolf Rotta, Steffen Büchner, Jörg Nolte
Operating Systems and Distributed Systems Group

Brandenburg University of Technology Cottbus

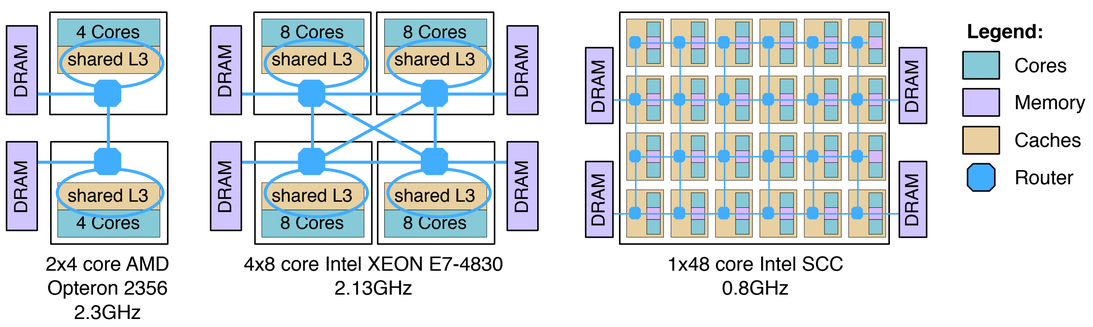
Introduction

- Many-core processors are hybrids of distributed and shared memory systems
- Middleware can combine advantages of messaging and shared memory
- Implemented prototype as C++ template framework

Why a middleware is necessary?

- High diversity between different many-core systems (e.g. network topology & capabilities), but relatively homogenous inside (single instruction set)

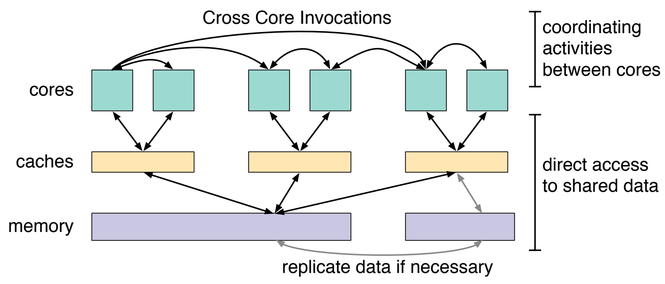
⇒ Use middleware to hide the diversity & achieve portable performance



What functionality should be provided by a middleware?

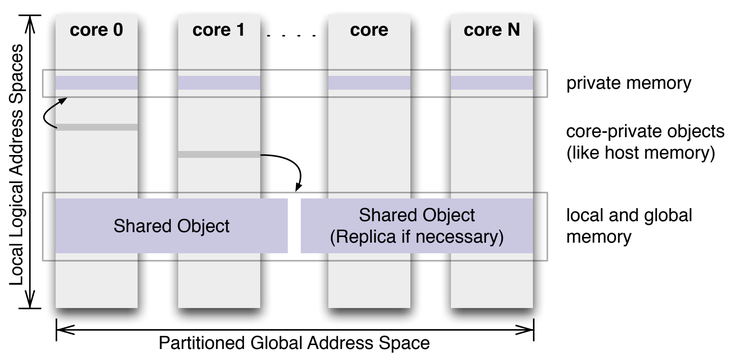
- Conventional shared memory programming: Scalability issues
- Conventional message passing: Large messages evict caches
- State of the art: Software components and object-orientation, bind components to cores (=Partitioned Global Address Space)
- Allocate shared objects in shared memory, pass references to shared objects and components by method calls

⇒ Implicit communication and execution across cores



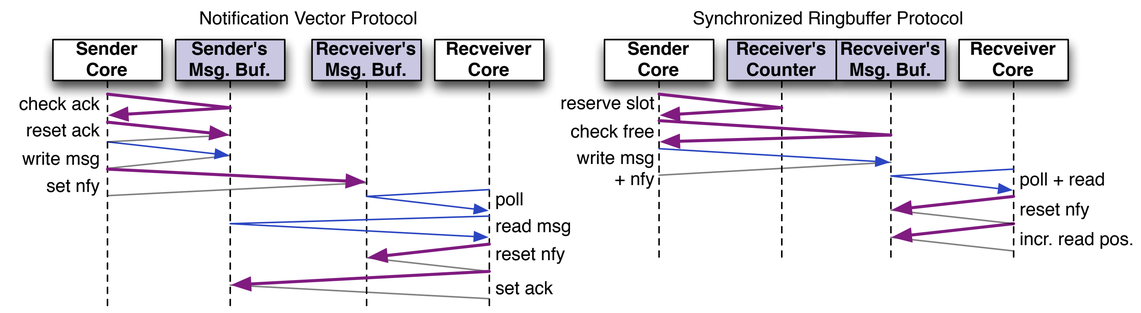
Shared Objects: Memory-efficient Sharing

- Access shared data directly through the cache hierarchy
- E.g. shared page tables, memory mapped files, large messages
- Use virtual memory to map physical memory to same logical address in all local address spaces, allocation is coordinated by CCIs, optional replication between distributed memory

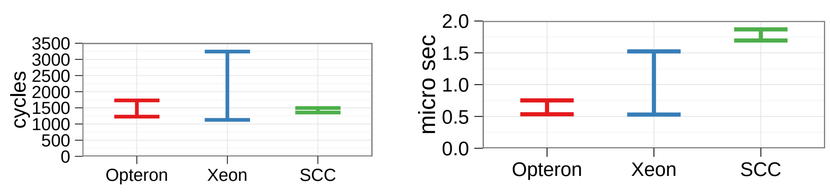


Cross Core Invocations: Coordination with Small Messages

- Initiate actions on other cores, offload tasks, wake up suspended activities
- E.g. offload tasks, query central services
- Lightweight Remote Method Invocation and Futures on top of small active messages (<200 bytes) and cooperative threads



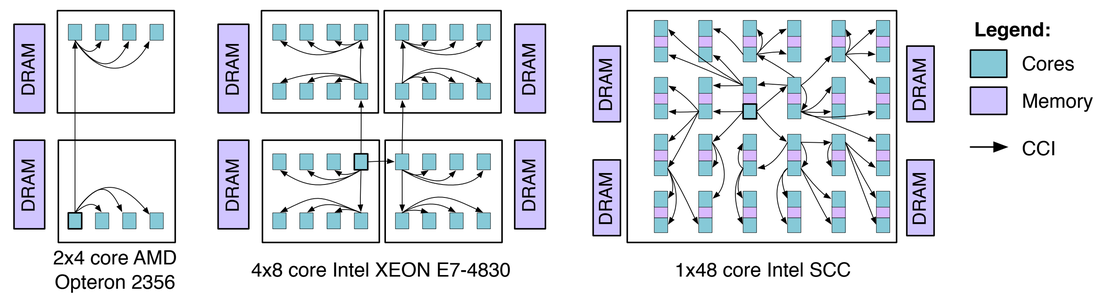
Example system-specific in-memory communication protocols.

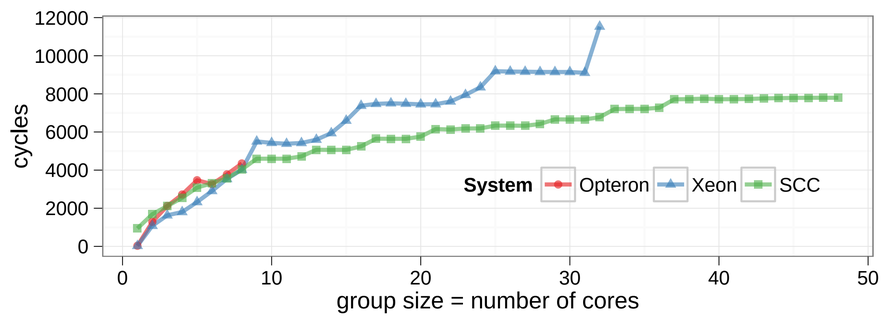


CCI roundtrip times ranging from nearest to farthest core.

Collective Operations: Group-Based Invocations

- One-sided initiation of activities on a group of cores, can wait for completion of all activities
- E.g. TLB shootdown, query distributed services
- One object per participating core, organized in multicast tree, propagate events as CCIs, collect&reduce results with Futures





Completion time of collective steps.

Conclusions

- Shared access to memory ⇒ no need for large messages
- Messages just for coordination ⇒ lightweight Cross Core Invocations
- Collective operations ⇒ scaleable coordination of many cores; Propagation mechanism can be adapted to network topology
- Small message exchange is critical for overall performance; In-memory communication ⇒ protocol overhead > network latency