The Journal of Systems and Software 97 (2014) 47–64

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Hybrid address spaces: A methodology for implementing scalable high-level programming models on non-coherent many-core architectures

Anastasios Papagiannis[a,*], Dimitrios S. Nikolopoulos[b]

[a] *Foundation for Research and Technology – Hellas, Institute of Computer Science (FORTH–ICS), Heraklion, Greece*
[b] *School of Electronics, Electrical Engineering and Computer Science, Queens University of Belfast, United Kingdom*

ARTICLE INFO

Article history:
Received 28 March 2013
Received in revised form 25 June 2014
Accepted 26 June 2014
Available online 5 July 2014

Keywords:
Parallel programming models
Runtime systems
Many-core processors

ABSTRACT

This paper introduces hybrid address spaces as a fundamental design methodology for implementing scalable runtime systems on many-core architectures without hardware support for cache coherence. We use hybrid address spaces for an implementation of MapReduce, a programming model for large-scale data processing, and the implementation of a remote memory access (RMA) model. Both implementations are available on the Intel SCC and are portable to similar architectures. We present the design and implementation of HyMR, a MapReduce runtime system whereby different stages and the synchronization operations between them alternate between a distributed memory address space and a shared memory address space, to improve performance and scalability. We compare HyMR to a reference implementation and we find that HyMR improves performance by a factor of 1.71× over a set of representative MapReduce benchmarks. We also compare HyMR with Phoenix++, a state-of-art implementation for systems with hardware-managed cache coherence in terms of scalability and sustained to peak data processing bandwidth, where HyMR demonstrates improvements of a factor of 3.1× and 3.2× respectively. We further evaluate our hybrid remote memory access (HyRMA) programming model and assess its performance to be superior of that of message passing.

1. Introduction

Many-core processors use diverging memory architectures. Processors designed for mainstream computing markets tend to use memory hierarchies with private multi-level caches per core and a hardware protocol to keep those caches coherent (Martin et al., 2012). This memory architecture resembles earlier shared-memory multi-processors from a programmer's standpoint. However, processors designed for more specialized markets, such as high performance computing and large-scale data processing, use memory hierarchies without a coherence protocol. Graphics Processing Units (GPUs) (Luebke et al., 2004), the Intel SCC (Mattson et al., 2010) the Cell BE processor (Kahle, 2005) and the experimental Runnemede prototype (Carter et al., 2013) are representative examples of non cache-coherent architectures. Programming a non-coherent architecture requires explicit communication between local address spaces, through message passing or Direct Memory Access (DMA). Explicit communication increases the programmer's burden, as it requires a high level of expertise in parallel programming and deep understanding of the memory hierarchy to master. However, explicit communication may also improve performance, particularly in applications with regular communication patterns. Programmers often opt for a programming model based on explicit communication even on cache-coherent many-core processors (Zimmer and Mueller, 2014; Tilera Processor Family, 2014), to exploit the topology of the interconnection network and minimize communication overhead. Runtime systems can ease the burden of programming with explicit communication to a certain extent by implementing high-level communication primitives and packaging them in user-level libraries (e.g. MPI). Alternatively, non-coherent architectures can be programmed with a high-level, shared address space model. In this case, the runtime system implements a virtual shared memory abstraction. Regardless of the choice of programming model, the runtime system is a critical component that largely defines performance, scalability and programmability.

* Corresponding author.
E-mail addresses: apapag@ics.forth.gr (A. Papagiannis), d.nikolopoulos@qub.ac.uk (D.S. Nikolopoulos).
URLs: http://www.ics.forth.gr/ apapag (A. Papagiannis), http://www.cs.qub.ac.uk/ D.Nikolopoulos (D.S. Nikolopoulos).

http://dx.doi.org/10.1016/j.jss.2014.06.058
0164-1212/© 2014 Elsevier Inc. All rights reserved.

Runtime systems for non cache-coherent architectures are currently implemented on top of distributed address spaces, typically using one address space per core. The runtime system itself implements all necessary inter-core communication operations for scheduling and synchronization, as well as all application-level communication through explicit message passing or DMAs. These operations flow either exclusively between local memories or between local memories and DRAM. This implementation paradigm has been used on the Cell BE processor, for implementing shared-memory programming models such as OpenMP (O'Brien et al., 2008), COMIC (Lee et al., 2008), Sequoia (Houston et al., 2008), and CellSs (Bellens et al., 2006) and the Intel SCC for the implementation of X10 (Chapman et al., 2011) and shared virtual memory models (Lankes et al., 2012). Intuitively, explicit communication in the runtime system yields a scalable implementation. In particular, explicit communication leverages on-chip data transfer paths and a scalable NoC interconnect for passing data between cores without paying the cost of off-chip memory accesses. This approach works particularly well for exchanges of messages that fit in on-chip local memories. However, this approach is not necessarily optimal in other cases. Applications often need to transfer large amounts of data between threads in a program with little or no processing on the data itself. If these streaming data transfers flow through the on-chip memory hierarchy, they will incur cache pollution, without offering an opportunity for data reuse. Such operations should be best left uncached to maximize performance. A shared, global address space model suits these operations best.

This paper introduces *hybrid address spaces*, as a fundamental design and implementation methodology for scalable runtime systems on non-coherent many-core architectures. The intuition behind hybrid address spaces is that a runtime system uses on-chip communication paths between private address spaces for small data transfers, such as those needed to exchange control data for scheduling, and off-chip communication paths through a shared address space, for large, streaming data transfers. To confirm our intuition, we present HyMR, an implementation of the MapReduce programming model (Dean and Ghemawat, 2008) on the Intel Single-Chip-Cloud Computer (Mattson et al., 2010). The MapReduce runtime implements a staged execution model. We show that while certain stages are best implemented with message passing over a distributed address space, other stages are best implemented with in-place memory copying in a single, global address space, or with a combination of distributed and shared address spaces. In demonstrating the concept of hybrid address spaces in runtime systems, we make several more contributions towards improving performance and scalability of MapReduce on non cache-coherent many-core architectures. These contributions include:

- software-controlled staged memory coherence to minimize the overhead of coherence maintenance;
- application-specific, scalable data splitters;
- scalable, interrupt-less work-stealing for non-coherent architectures using exclusively on-chip communication;
- a new implementation of scalable on-chip barrier algorithms for non-coherent many-core processors;
- a new mechanism to enable fast access from a core to the private memory of another core on-chip, which accelerates global exchange operations;
- a parallel sorting algorithm that avoids synchronization between stages and executes critical communication paths using on-chip shared memory.

Our implementation of HyMR provides design guidelines for latency and throughput critical runtime system operations that are common to many, if not all, programming models. These include scheduling and load balancing, data distribution, point-to-point and group communication operations.

We compare HyMR to a reference runtime system implemented using exclusively message passing. HyMR outperforms the baseline in all tests. We also compare HyMR with Phoenix++, a state-of-art MapReduce implementation for hardware-managed cache-coherence systems (Talbot et al., 2011). HyMR achieves, on average, 3.1× improvement in speedup and 3.2× improvement of bandwidth efficiency compared to Phoenix++, on the same number of cores.

To further demonstrate hybrid address spaces as a viable methodology for implementing parallel programming models, we have also developed a hybrid remote memory access (HyRMA) programming model, which leverages message passing for on-chip, latency-sensitive data one-way transfers and global shared memory for one-way bulk data transfers. We demonstrate HyRMA with a representative stencil code, the Jacobi method. HyRMA improves performance by a factor of 2.41× using 48 cores, compared to a pure message passing approach.

The rest of this paper is organized as follows: Section 2 provides background on MapReduce and the Intel SCC processor. Section 3 presents the design and implementation of DiMR, a reference implementation of the MapReduce runtime for SCC processor, which uses exclusively distributed address spaces. Section 4 presents the design and implementation of HyMR. Section 5 presents our experimental analysis and results. Section 6 presents the implementation and experimental analysis of HyRMA. Section 7 discusses related work and Section 8 concludes the paper.

2. Background

Hardware support for cache coherence on a processor with many cores increases complexity and power (Kavadias et al., 2010). Although many efforts attempt to address the scaling and power limitations of cache coherence on systems with many cores (Martin et al., 2012), several vendors of many-core processors opt for a non cache-coherent architecture. On such an architecture, a programmer writes parallel code using either explicit communication mechanisms or a shared virtual memory layer implemented in software. In this section we provide background on non cache-coherent many-core processors and programming models providing a shared memory abstraction on such processors. We discuss in more detail the architecture of the Intel Single-Chip-Cloud Computer (SCC), a processor prototyped to explore the scalability, programmability and power-efficiency of non-coherent architectures. We use the SCC as an implementation vehicle for implementing scalable runtime systems with hybrid address spaces. The use of SCC is by no means limiting our study: our runtime system design and implementation techniques presented later in this paper generalize to any non-coherent many-core processor with programmable memory mapping/translation tables and a mechanism for explicit on-chip communication between cores. We conclude this section by providing background on MapReduce, a parallel programming model for large-scale data processing, inspired by functional languages.

2.1. Intel Single-Chip-Cloud Computer (SCC)

The Intel SCC[1] (Howard et al., 2010) (Fig. 1) is a many-core processor with 24 tiles and two IA cores per tile. The tiles are organized in a 4×6 mesh network with 256 GB/s bisection bandwidth. The processor has four integrated DDR3 memory controllers, one for

[1] The SCC is not a stand-alone computer thus to get it running, a management PC (MCPC) needs to be used. The SCC connects to the MCPC through external PCIe.