Microprocessors and Microsystems 43 (2016) 95–103

Contents lists available at ScienceDirect

Microprocessors and Microsystems

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

Parallel programming model for the Epiphany many-core coprocessor using threaded MPI

James A. Ross[a,*], David A. Richie[b], Song J. Park[a], Dale R. Shires[a]

[a] U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, United States
[b] Brown Deer Technology, Forest Hill, MD, United States

ARTICLE INFO

Article history:
Received 7 July 2015
Revised 4 February 2016
Accepted 5 February 2016
Available online 23 February 2016

Keywords:
2D RISC array
MPI
NoC
Many-core
Adapteva Epiphany
Energy efficiency

ABSTRACT

The Adapteva Epiphany many-core architecture comprises a 2D tiled mesh Network-on-Chip (NoC) of low-power RISC cores with minimal uncore functionality. It offers high computational energy efficiency for both integer and floating point calculations as well as parallel scalability. Yet despite the interesting architectural features, a compelling programming model has not been presented to date. This paper demonstrates an efficient parallel programming model for the Epiphany architecture based on the Message Passing Interface (MPI) standard. Using MPI exploits the similarities between the Epiphany architecture and a conventional parallel distributed cluster of serial cores. Our approach enables MPI codes to execute on the RISC array processor with little modification and achieve high performance. We report benchmark results for the threaded MPI implementation of four algorithms (dense matrix–matrix multiplication, *N*-body particle interaction, five-point 2D stencil update, and 2D FFT) and highlight the importance of fast inter-core communication for the architecture.

Published by Elsevier B.V.

1. Introduction

The emergence of a wide range of parallel processor architectures continues to present the challenge of identifying an effective programming model that provides access to the capabilities of the architecture while simultaneously providing the programmer with familiar, if not standardized, semantics and syntax. The programmer is frequently left with the choice of using a non-standard programming model specific to the architecture or a standardized programming model that yields poor control and performance.

The Adapteva Epiphany MIMD architecture is a scalable 2D array of RISC cores with minimal uncore functionality connected with a fast 2D mesh Network-on-Chip (NoC) [1]. Processors based on this architecture exhibit good energy efficiency and scalability via the 2D mesh network, but require a suitable programming model to fully exploit the architecture. The 16-core Epiphany III [2] coprocessor has been integrated into the Parallella minicomputer platform [3] where the RISC array is supported by a dual-core ARM CPU and asymmetric shared-memory access to off-chip global memory. Fig. 1 shows the high-level architectural features of the coprocessor. Each of the 16 Epiphany III mesh nodes contains 32KB of shared local memory (used for both program instructions and data), a mesh network interface, a dual-channel DMA engine, and a RISC CPU core. Each RISC CPU core contains a 64-word register file, sequencer, interrupt handler, arithmetic logic unit, and a floating point unit. Each processor tile is very small at 0.5 mm^2 on the 65 nm process and 0.128 mm^2 on the 28 nm process. Peak single-precision performance for the Epiphany III is 19.2 GFLOPS with a 600 MHz clock. Fabricated on the 65 nm process, the Epiphany III consumes 594 mW for an energy efficiency of 32.3 GFLOPS/W (Olofsson, personal communication). The 64-core Epiphany IV, fabricated on the 28 nm process, has demonstrated energy efficiency exceeding 50 GFLOPS/W [4].

The raw performance of currently available Epiphany coprocessors is relatively low compared to modern high-performance CPUs and GPUs; however, the Epiphany architecture provides greater energy efficiency and is designed to be highly scalable. The published architecture road map specifies a scale-out of the architecture to exceed 1000 cores in the near future and, shortly thereafter, tens of thousands of cores with an energy efficiency approaching 1 TFLOPS/W. Within this context of a highly scalable architecture with high energy efficiency, we view it as a competitive processor technology comparable to GPUs and other coprocessors.

While architectural energy efficiency is important, achievable performance with a compelling programming model is equally, if not more, important. Key to performance with the Epiphany architecture is data re-use, requiring precise control of inter-core

* Corresponding author. Tel.: +1 4102789556.
E-mail addresses: james.a.ross176.civ@mail.mil, james.a.ross@gmail.com (J.A. Ross), drichie@browndeertechnology.com (D.A. Richie), song.j.park.civ@mail.mil (S.J. Park), dale.r.shires.civ@mail.mil (D.R. Shires).

http://dx.doi.org/10.1016/j.micpro.2016.02.006
0141-9331/Published by Elsevier B.V.

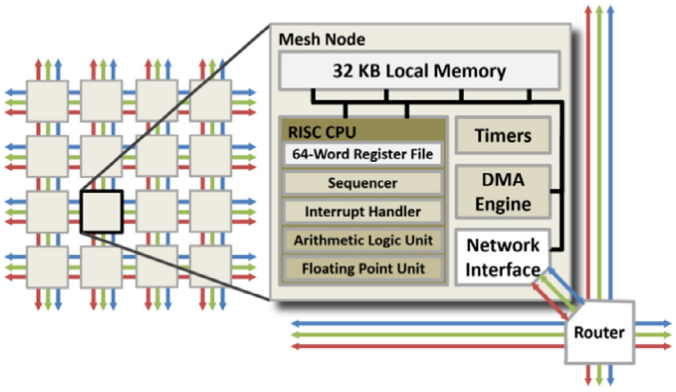

Fig. 1. Adapteva Epiphany III architecture diagram.

communication since the architecture does not provide a hardware cache at any level. The cores can access off-chip mapped memory with a significant performance penalty in both latency and bandwidth relative to accessing neighboring core memory.

When developing parallel applications, the parallel programming model and API must match the architecture lest it becomes overly complicated or unable to achieve good performance. By leveraging the standard MPI programming model, Epiphany software developers inherit a multi-decadal legacy of refined software design and domain decomposition considerations. For the Epiphany architecture, an MPI programming model is a better choice than OpenCL, which primarily targets GPUs. MPI is also superior to OpenMP since shared memory exhibits significant NUMA issues. Neither OpenCL nor OpenMP provide a mechanism for explicitly controlling inter-core data movement, which is critical to achieving high performance for anything but trivially parallel applications on this processor. Further discussion of OpenCL is found in Section 3.3. Other programming models such as partitioned global address space (PGAS) using Unified Parallel C (UPC) may have merit with the Epiphany architecture; however, it requires an extension to ANSI C and removes some explicit control from the programmer. The SHMEM library, which does not require compiler support, is a good candidate for future development on the Epiphany architecture.

In this paper, we demonstrate that threaded MPI exhibits the highest performance reported to date using a standard parallel API on the Epiphany architecture. Threaded MPI allows algorithm design to closely follow the methods for distributed parallel processors, and in some cases the code can be re-used with minimal modifications. Our previous demonstration of threaded MPI for dense matrix–matrix multiplication on the Epiphany architecture [5] has been improved by optimizing the MPI communication and algorithm computational performance. This paper is an extension to the publication in [6], where additions include a discussion of the platform roofline model, application arithmetic intensities, the inter-core communication patterns, a detailed description of the critical MPI_Sendrecv_replace communication routine, additional analysis of application and threaded MPI communication performance on the platform, and more reference material. Benchmarks for on-chip performance demonstrate that threaded MPI can be broadly applied to different algorithms. The peak performance achieved for each benchmark compares favorably with results reported for related benchmarks on other coprocessors, including the Intel Xeon Phi and Teraflops Research Chip.

2. Threaded MPI

Threaded MPI was developed to provide an extremely lightweight implementation of MPI appropriate for threads executing within the restricted context of the Epiphany RISC cores. Threaded MPI enables the use of a standard parallel programming API to achieve high performance and platform efficiency. The MPI programming model is well suited to the Epiphany architecture. The use of MPI for programming the Epiphany architecture was first suggested with a simple proof-of-concept demonstration in 2013 [7], and it is somewhat surprising that this line of research is only now being more thoroughly explored on this platform. Examining a threaded implementation of MPI for multi-core CPUs is not new [8,9]. However, previous investigations did not address the significant architecture constraints found with Epiphany and employed a more conventional design. As a result, existing MPI implementations provide no possibility for porting, nor do they provide any guide for the design of a threaded implementation targeting this class of architecture.

Threaded MPI is distinguished from conventional MPI implementations by two critical differences directly related to the architecture. The Epiphany device must be accessed as a coprocessor and each core executes threads within a highly constrained set of resources. As a result, the cores are not capable of supporting a full process image or program in the conventional sense, and therefore the conventional MPI model of associating MPI processes to concurrently executing programs is also not possible. Instead, coprocessor offload semantics must be used to launch concurrent threads that will then employ conventional MPI semantics for inter-thread communication.

The practical consequence is that rather than launching an MPI job from the command line, a host program executing on the platform CPU initiates the parallel MPI code using a functional call; the mpiexec command is replaced with an analogous function call, coprthr_mpiexec (int device, int np, void* args, size_t args_sz, int flags). This has the advantage of localizing the parallelism to a fork-join model within a larger application that may otherwise execute on the platform CPU, and multiple coprthr_mpiexec calls may be made from within the same application. From the host application executing on the platform CPU, explicit device control and distributed memory management tasks must be used to coordinate execution with the Epiphany coprocessor at a higher level. These host routines are separate from the MPI programming model used to program the coprocessor itself. The only practical consequence and distinction with MPI code written for Epiphany, compared with a conventional distributed cluster, is that the main() routine of the MPI code must be transformed into a thread function and employ Pthread semantics for passing in arguments. Beyond this, no change in MPI syntax or semantics is required.

The more serious challenge that directly impacts both the performance and the range of application of a threaded MPI implementation is the significantly limited amount of local memory per core. A conventional MPI implementation relies upon large buffers for message queues tuned for the specific host and network parameters. Here, no more than 32KB are available per core for supporting program instructions, local storage, and message buffers. Whereas, the cores do have access to a much larger shared memory region in global DRAM (on the order of 32MB), the cost associated with accessing this global memory, as compared to the extremely low latency of local memory, prevents the use of large MPI buffers in global memory. Such a design would compromise the known requirements for achieving good performance and therefore would not be efficient.

A minimal subset of the MPI standard, shown in Table 1, was implemented at the time of this work. Support was provided for basic initialization, the creation of Cartesian topologies, blocking send/receive pairs, and a combined blocking send/receive/replace call. These calls are sufficient for nontrivial experiments to test the effectiveness of the overall approach and implementation. For example, these routines provide the minimal set required to implement primitive MPI send and receive tests as well as porting a