2015

Threaded MPI programming model for the Epiphany RISC array processor

David Richie
Brown Deer Technology, MD

James Ross
Engility Corporation, MD

Song Park
U.S. Army Research Laboratory, MD

Dale Shires
U.S. Army Research Laboratory, MD

Journal of Computational Science 9 (2015) 94–100

Contents lists available at ScienceDirect

Journal of Computational Science

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

Threaded MPI programming model for the Epiphany RISC array processor

David Richie [a,*], James Ross [b], Song Park [c], Dale Shires [c]

[a] Brown Deer Technology, MD, USA
[b] Engility Corporation, MD, USA
[c] U.S. Army Research Laboratory, MD, USA

ARTICLE INFO

Article history:
Available online 17 April 2015

Keywords:
2D RISC array
Threaded MPI
Adapteva Epiphany
Parallella
Energy efficiency

ABSTRACT

The low-power Adapteva Epiphany RISC array processor offers high computational energy-efficiency and parallel scalability. However, extracting performance with a standard parallel programming model remains a great challenge. We present an effective programming model for the Epiphany architecture based on the Message Passing Interface (MPI) standard adapted for coprocessor offload. Using MPI exploits the similarities between the Epiphany architecture and a networked parallel distributed cluster. Furthermore, our approach enables codes written with MPI to execute on the RISC array processor with little modification. We present experimental results for matrix–matrix multiplication using MPI and highlight the importance of fast inter-core data transfers. Using MPI we demonstrate an on-chip performance of 9.1 GFLOPS with an efficiency of 15.3 GFLOPS/W. Threaded MPI exhibits the highest performance reported for the Epiphany architecture using a standard parallel programming model.

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 often 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 RISC array architecture [1] is designed to be a scalable 2D array of low-power RISC cores with minimal uncore functionality supported by an on-chip 2D mesh network for fast inter-core communication. The Epiphany architecture is scalable to 4096 cores and represents an example of an architecture designed for power-efficiency at extreme on-chip core counts. Processors based on this architecture exhibit good performance/power metrics [2] and scalability via 2D mesh network [3,4], but require a suitable programming model to fully exploit the architecture. A 16-core Epiphany III coprocessor [5] has been integrated into the Parallella mini-computer platform [6] where the RISC array is supported by a dual-core ARM CPU and asymmetric shared-memory access to off-chip global memory.

RISC array processors such as those based on the Epiphany architecture may offer significant computational power efficiency in the near future with requirements in increased core counts, including long-term plans for exascale platforms. The power efficiency of the Epiphany architecture has been specifically identified as both a guide and prospective architecture for such platforms [7]. The Epiphany IV processor has a performance efficiency of 50 GFLOPS/W [2] making it one of the most efficient parallel processors based on general-purpose cores and satisfying the threshold for exascale computing with a power budget of 20 megawatts [8]. This architecture has characteristics consistent with future processor predictions arguing hundreds [9] and thousands [10,11] of cores on a chip.

The Epiphany architecture is also interesting from a computer architecture perspective. The 2D mesh topology of the RISC array network creates a device-scale architecture that resembles a classic parallel distributed cluster of serial processors, where the Message Passing Interface (MPI) standard remains the programming model of choice. It is not possible, nor would it be efficient, to run a full MPI implementation on an Epiphany RISC array. However, the conceptual programming model is very well suited to the task if the implementation is recast into a restricted threaded form. Managing inter-core communication is critical to achieving good performance

* Corresponding author.
E-mail addresses: drichie@browndeertechnology.com (D. Richie), james.ross@engilitycorp.com (J. Ross), song.j.park.civ@mail.mil (S. Park), dale.r.shires.civ@mail.mil (D. Shires).

http://dx.doi.org/10.1016/j.jocs.2015.04.023
1877-7503/

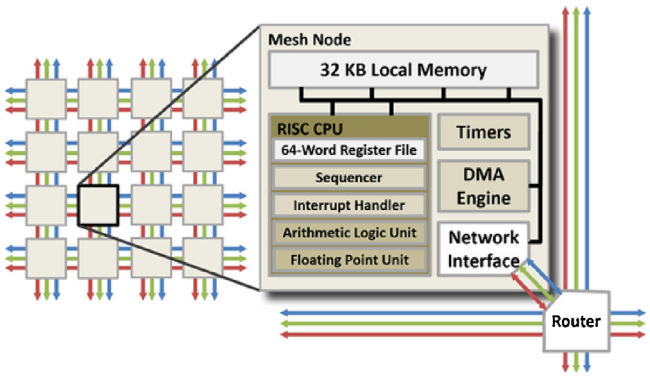

Fig. 1. The Epiphany RISC array architecture. RISC cores are connected through a point-to-point network for signaling and data transfer. Communication latency between cores is low, but the amount of addressable data contained on a mesh node is low (32 KB). The three networks shown above handle local read transactions, local write transactions, and off-chip memory transactions.

with the Epiphany architecture since this was a central element in the design of the architecture. Therefore, it is interesting to explore the utility of MPI for programming on-chip parallelism.

We present an investigation into the use of a lightweight threaded implementation of MPI for the Epiphany architecture. The investigation is motivated by the potential benefits of a parallel programming model for a RISC array processor capable of extracting high performance for real applications using a familiar and pervasive API. The use of MPI would also allow reuse of existing code, albeit with small modifications. Even for newly developed applications, the use of a proven API for parallel programming would provide substantial benefits for programmers. If successful, this approach would favorably resolve one of the most significant challenges for programming RISC arrays processors like those based on the Epiphany architecture.

Our main contributions are as follows: we present a novel threaded MPI programming model for the Epiphany architecture, we provide promising performance results for a matrix–matrix multiply algorithm using an essentially unmodified MPI code, and we demonstrate that the threaded MPI programming model extends to support larger problem sizes that exceed the available local memory. An outline of the remainder of the paper is as follows. Section 2 describes the relevant features of the Epiphany RISC array architecture. Section 3 explains the design and implementation of a threaded MPI programming model. Section 4 describes the reuse of an existing MPI implementation for matrix–matrix multiplication for the Epiphany architecture. Section 5 presents a discussion of benchmark results. Section 6 highlights related prior work using the Epiphany processor. Section 7 provides concluding observations.

2. Background

The Epiphany architecture is based on a 2D array of low-power 32-bit RISC cores, each with 32 KB of fast local memory and a robust mesh network for fast inter-core communication. The fully memory-mapped architecture allows shared memory access to global off-chip memory and shared non-uniform memory access to the local memory of each core. A block diagram of the Epiphany architecture is shown in Fig. 1. In many ways the architecture resembles a chip-scale instantiation of a traditional cluster with a 2D mesh network topology. This similarity makes it interesting to consider a parallel programming model for Epiphany based on MPI. MPI has proven to be the most pervasive parallel programming API for high-performance computing [12]. In simplest terms, the protocol supports semantics of one- and two-sided communication between parallel processes allowing the exchange of data. The typical application of MPI employs a parallel data decomposition of a problem over processes executed in parallel with MPI supporting the inter-process communication. Common communication patterns are neighbor exchange, all-to-all broadcast, and reduction operations. MPI generally relies upon a robust inter-node communication network to provide for high-bandwidth data transfers. MPI supports inter- and intra-node parallelism typical of parallel systems with multi-core processors.

However, the differences between the Epiphany architecture and a networked parallel distributed cluster of processors present challenges in developing an effective MPI implementation. Treating each core in analogy to a node in a cluster encounters the challenge of the greatly limited memory available on-chip and the lack of a hardware cache. Whereas, a typical cluster node may contain on the order of gigabytes of memory, each core in the RISC array has only 32 KB that must be used for both instructions and local data storage. Although each core has shared memory access to off-chip global memory, this access is significantly slower than local memory. For this reason, off-chip global memory cannot be used for message buffers, and as a consequence, a conventional MPI implementation is not possible since typical buffers on the order of megabytes are often reserved for message transactions. Further, the limited on-chip resources prevent the execution of a full process image like a conventional node in a cluster, and the execution of parallel tasks must be offloaded to the Epiphany coprocessor as lightweight threads. The architectural feature that must be exploited in any implementation is the extremely low latency access to local memory between cores supported by the 2D mesh network.

3. Threaded MPI

3.1. Design

The following design objectives were identified in implementing MPI for the Epiphany architecture. First, the implementation should maintain the precise syntax of MPI, and to the greatest extent possible, the semantics should be maintained with minimal restrictions and without extensions. Second, any implementation must be efficient and capable of accessing the performance and capabilities of the processor architecture. The unique architectural features, different from that of a networked cluster, must be exploited and the limitations must be accounted for in a way that minimizes the impact on performance. Finally, the initial design should be pragmatically focused on the most essential MPI calls necessary for real demonstrations.

A key architectural feature exploited in the design is the fast on-chip mesh network that enables low-latency access to the local memory of any core through a fully memory-mapped address space. Enabling efficient on-chip data transfers and data reuse while minimizing access to global DRAM is essential to achieving good performance. In addition, the platform integration of the RISC array as a coprocessor with shared access to the global memory of the ARM CPU host is used to enable efficient coprocessor offload mechanisms. Overall, there were two significant challenges that proved to drive the design and implementation of threaded MPI for the architecture.

First, the RISC array must be used as a coprocessor with compute offload supported by the ARM CPU host. Furthermore, the Epiphany cores cannot support the efficient execution of a full process image or program that is typically used as the mechanism for parallel decomposition in conventional MPI implementations. Whereas, a conventional MPI program is initiated by the mpiexec command that launches multiple processes on a parallel platform, here we must have a single host program running on an ARM CPU

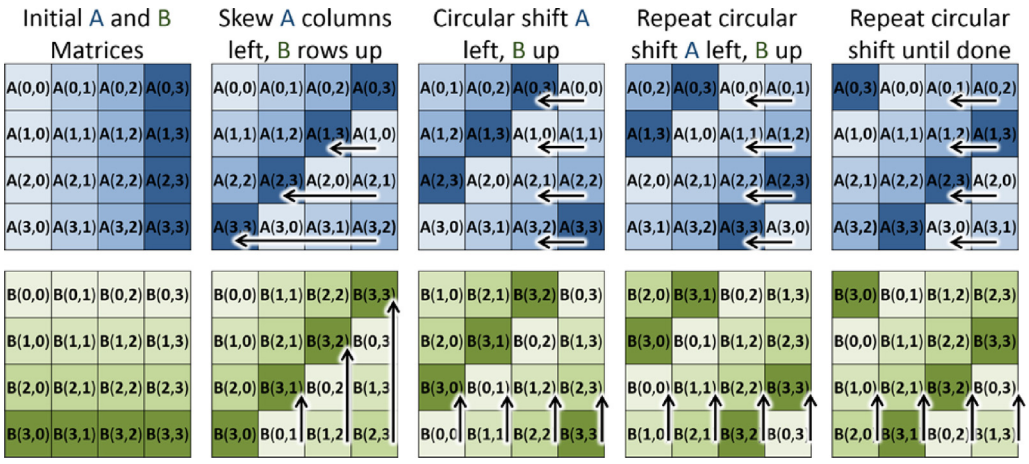

Fig. 2. The 2D mesh network topology communication patterns for submatrix skewing and shifting. A submatrix–submatrix mutliplication occurs after each communication step. For the Epiphany III processor, this figure represents the full communication pattern between the 16 cores on the device although the communication pattern can be applied generally to larger or smaller square arrays of cores. An additional communication step is needed to restore the shifted and skewed matrices if desired, but this is unnecessary since a copy of the *A* and *B* matrices remains within shared device memory.

initiate parallel MPI tasks as parallel threads executing within a restricted set of resources. The conventional mpiexec command must be transformed into a functional call executed by the CPU host program. The parallelism becomes localized within the larger application, with an execution model resembling that of a fork-join semantic similar to other APIs such as OpenMP, CUDA, or OpenCL. However, unlike these other APIs, here the threads will execute with explicit message passing support. This model has an advantageous consequence whereby the parallel context becomes localized and modularized, for example multiple mpiexec calls can be made within a single application utilizing different parallel decompositions.

The second and more serious challenge that will directly impact both 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 32 KB 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 32 MB), 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 could not be efficient. This constraint would at first appear to preclude any implementation of MPI for programming the architecture. However, we will show in the following that using a strategy of zero–copy or small buffered communication is sufficient to implement MPI semantics with good overall performance as a parallel programming model for on-chip inter-core communication.

The overarching issues stem from the fact that the architectural context of the Epiphany RISC array architecture differs substantially from that of a networked cluster of processors for which MPI is primarily designed and implemented. Examining a threaded implementation of MPI for multi-core CPUs is not new [13], however, previous investigations did not address the significant architecture constraints found with Epiphany and employed a more conventional design. As a result of these architectural differences, existing MPI implementations provide no possibility for porting, nor do they provide any guide for the design of a threaded implementation. This did not, however, prevent an alternate approach and implementation of MPI targeting the architecture.

3.2. Implementation

A threaded MPI implementation has been developed based on the Epiphany support provided by the COPRTHR SDK [14]. The COPRTHR run-time support for coprocessor device management and offloading threads to the coprocessor, as well as the clcc thread compiler were used to develop a thin-layer implementing threaded MPI as part of the COPRTHR software stack. Existing support for distributed memory management is used to manage data transfers between the host CPU and Epiphany coprocessor. The threaded MPI implementation targets the Epiphany architecture and has been functionally tested on a production 16-core Epiphany III processor and an engineering sample of a 64-core Epiphany IV processor.

In analogy to the mpiexec command, the coprthr_mpiexec call was developed to launch parallel threads on the coprocessor, and relies on the existing Pthreads-like coprthr_ncreate call already provided by the COPRTHR run-time. The coprthr_mpiexec call is the semantic equivalent to the conventional mpiexec command, used here to launch parallel threads as opposed to parallel processes. Here, the MPI thread function plays the role of the conventional MPI main program. The only distinction is the trivial requirement of extracting thread arguments at the beginning of the thread function using the Pthread semantic of passing a pointer to a struct data type containing the arguments. After this is done, identical MPI code may be used to implement a parallel algorithm.

The MPI calls listed in Table 1 have been implemented for the Epiphany architecture. Support is 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 subset of pre-existing algorithms implemented with MPI.

The MPI_Send and MPI_Recv calls are implemented as tightly coupled zero–copy communication routines with very low-latency inter-core synchronization. The Epiphany architecture supports direct read and write operations from one core to the local memory of another, where write transactions have priority over read transactions. The protocol chosen to implement a send–receive pair has the receiving thread setting up and enabling the transaction, and the sending thread driving the data transfer using a direct write

Table 1
List of MPI calls used within the matrix–matrix multiply application code.

MPI Call	Comments
MPI_Init	Initialization, called once per thread
MPI_Finalize	Finalization, called once per thread
MPI_Comm_size	Get number of threads in the group
MPI_Comm_rank	Get rank of the current thread in the group
MPI_Cart_create	Create 1D and 2D communicator topologies
MPI_Comm_free	Free communicator
MPI_Cart_coords	Determines thread coordinates in topology
MPI_Cart_shift	Returns the shifted source and destination ranks in topology
MPI_Send	Blocking send, zero–copy implementation
MPI_Recv	Blocking receive, zero–copy implementation
MPI_Sendrecv_replace	Blocking send/ receive/replace, buffered implementation
MPI_Bcast	Broadcast, zero–copy implementation

operation targeting the local memory of the receiving core. This protocol is optimized for the Epiphany architecture.

The symmetric MPI_Sendrecv_replace call cannot be implemented using a zero–copy design, but instead requires a buffered transaction. The challenge here is that the local memory per core is highly constrained, making the dedication of a large buffer for transactions extremely costly, if at all possible, and prohibitive in typical use-cases. For this reason a protocol using a small buffer is employed and message transfers are transparently broken up into multiple buffered transactions. The optimization of this protocol is critical for performance. We demonstrate in the following that the protocol does in fact achieve excellent performance in a matrix–matrix multiplication algorithm that relies heavily upon the use of this call. The effect of tuning the size of this small MPI buffer will be examined in detail.

4. Matrix–matrix multiplication application

Multiplication of matrices is a central building block in many scientific applications. In order to investigate the utility of MPI as a programming model for the Epiphany architecture, Cannon's algorithm [15] for matrix–matrix multiplication based on an existing MPI implementation [16] was employed. Cannon's algorithm exemplifies the use of 2D parallel decomposition to effectively exploit this type of parallel architecture. The algorithm decomposes a square matrix–matrix multiplication problem ($C = A \times B$) across an N-by-N collection of processing elements. Submatrices are shared between neighboring processing elements after each submatrix–submatrix multiplication. As illustrated in Fig. 2, the communication pattern begins by skewing the columns of matrix A left and the rows of B upward within the 2D mesh network topology.

The advantage of this algorithm is its fixed storage size and regular communication pattern suitable for the Epiphany architecture. Because the A and B matrices are copies from the shared memory, there is no need to restore the original configuration of the matrices after the calculation is complete. There are as many communication and multiplication steps as there are rows or columns of cores. For problems that cannot fit entirely within the available Epiphany core-local memory, additional outer loops are required. The matrix–matrix multiplication operation is self-similar at multiple scales, therefore the smaller matrix calculation is analogous to a standard matrix–matrix multiply operation. For larger problems that cannot fit within the available on-chip memory, there is a performance penalty for off-chip communication.

We transformed the existing MPI code into a thread function to use the threaded MPI implementation for Epiphany. The body of the original MPI code remains essentially unmodified. The most significant change is the small addition of code at the beginning of the thread function to extract arguments passed in using Pthreads-style semantics. The thread function is then compiled using the COPRTHR clcc thread compiler targeting Epiphany and linked with the COPRTHR MPI library. A small host program was used to manage the coprocessor device and distributed memory transfers as well as to make the coprthr_mpiexec call to launch the parallel task. Moreover, the original MPI Cannon code was extended to support larger matrices by cycling submatrices through global memory. It is important to note that even these further modifications did not alter the original low-level MPI code implementing the on-chip algorithm.

5. Experimental analysis

The 16-core Epiphany III coprocessor available with the Parallella platform was used for development and benchmarking. The Epiphany III operates at 600 MHz and has a peak performance of 19.2 GFLOPS. Cannon's algorithm was benchmarked using the threaded MPI implementation for small matrices that fit entirely in the on-chip local memory distributed over the 16 cores. This allowed matrices in the range of 32×32–128×128 to be treated for on-chip calculations. Overall on-chip performance is shown in Fig. 3 for various MPI buffer sizes.

The MPI buffer size was carefully studied for different problem sizes by measuring performance for MPI buffer sizes ranging from 64 to 1536 bytes. The effect of increasing the relatively small MPI buffer, used internally for the buffered MPI_Sendrecv_replace call, can be seen with the increasing performance shown in Fig. 3. The results indicate that the MPI communication overhead is nearly converged with an MPI buffer size of 512 bytes. This is an important observation since the highly constrained per-core local memory makes the allocation of a large buffer for MPI communication costly, if not prohibitive. The experimental results reveal that despite these resource constraints, an effective mechanism for buffered transfers can be implemented without a large MPI buffer. This is possible since, as compared to a traditional MPI implementation, the threaded MPI benefits from extremely low-latency communication between parallel threads.

In order to better understand the relative costs impacting the overall performance, timing measurements were made to study the distribution of time expended for floating-point computation, inter-core data transfer, and MPI communication overhead. The floating-point computation time is defined as the time expended within the innermost loop structure where each core sums the contributions to the product matrix over values of matrix A and B stored in local memory. An effective breakdown of the time per matrix–matrix multiplication and on-chip data transfer are presented in Fig. 4a for matrices with size ranging from 32×32 to 128×128 with an MPI temporary buffer of 512 bytes. The percentage of data transfer in relation to total time is decreased as the matrix size is increased. Fig. 4b shows the total on-chip bandwidth observed for the inter-core data transfer. In addition, the overhead of the MPI calls excluding data transfer time was calculated to be

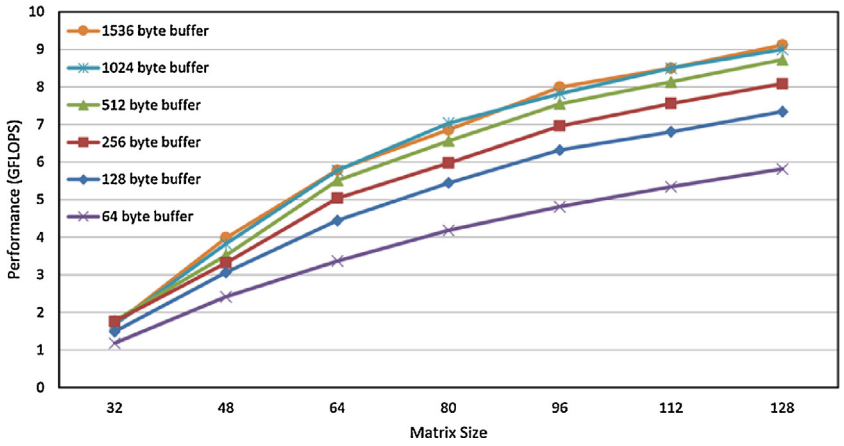

Fig. 3. The measured performance of matrix–matrix multiplication for an Epiphany III processor while varying the MPI buffer size and the on-chip problem size. Increasing the MPI buffer beyond 512 bytes generally has very marginal performance gains. An MPI buffer size of 256 bytes is sufficient to hold the entire 8×8 submatrix in the 32×32 matrix–matrix multiplication. Using more than a 1536 byte buffer on the 128×128 problem caused the instruction and data storage requirement to exceed the 32 KB per core limit.

low and on the order of 15%. These results demonstrate the efficiency of the threaded MPI implementation for programming the Epiphany architecture, since it was not known whether the overhead of the MPI implementation would dominate execution time as a bottleneck in performance. This is demonstrated not to be the case.

Benchmarks were extended to include larger matrices (up to 1536×1536) that fit within the 32 MB shared off-chip memory as well as even larger matrices (up to 6144×6144) requiring nearly all of the available platform memory for storage. For problem sizes that exceed the available Epiphany local memory, the off-chip communication to the 32 MB host-coprocessor shared memory approaches a bandwidth saturation point, limiting the performance to approximately 2.7 GFLOPS, which excludes the overhead of copying memory from the original host location. Fig. 5 shows the performance for various problem sizes using a 512 byte MPI buffer. The largest problem that fits within the shared local memory on-chip is 128×128 while the 32 MB shared memory can hold a 1536×1536 problem. For computations that exceed the shared memory, an additional host-side copy routine moves large submatrix data blocks into the 32 MB buffer, thus limiting the problem size to the available address space of the host system. Using this technique which we describe as host-interactive, problem sizes of up to 6144×6144 were executed and achieved approximately 2.35 GFLOPS including the overhead of copying the memory from the host memory to the shared memory. Larger problem sizes are limited by platform memory.

Whereas, the total performance demonstrated with this benchmark is less than that of other processors, for example high-end CPUs and GPUs, it should be compared within the context of the superior power efficiency of the Epiphany architecture. The Epiphany III processor used here for benchmarking consumes approximately 594 mW. Therefore, the demonstrated performance using threaded MPI achieving 9.1 GFLOPS corresponds to 15.3 GFLOPS/W. In addition, the Epiphany architecture was designed to scale to 1000's of cores using the same 2D RISC array topology present in the current production processors. As such, the results presented here are a positive indication of the potential performance of future processors based on this architecture at the extreme core counts on the current roadmap. The possibility of achieving this level of performance using a familiar and conventional parallel programming model also has strong implications at the extreme scaling of the architecture.

The results presented here may also be compared to the work in [7] where an Epiphany IV processor was studied using hand-coded assembly, special data alignment, and custom inter-core communication code, achieving 16.2 GFLOPS for a 4×4 core array. By contrast, using ordinary C code with MPI, we achieve 56% of this performance with 9.1 GFLOPS. Furthermore, additional optimizations are available for future work such as overlapping computation and communication without going outside the MPI standard, using MPI one-sided communication calls. Achieving this level of performance using a standard parallel programming model, and even re-using MPI code originally written for a parallel cluster, has

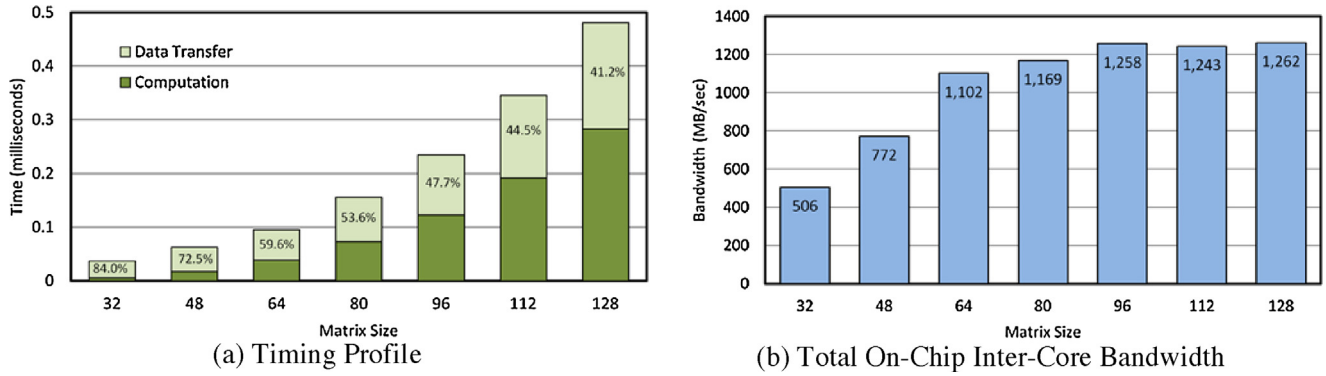

Fig. 4. (a) Breakdown of time spent for submatrix multiplication (compute) vs. data transfer for various matrix sizes. (b) The total on-chip bi-directional inter-core bandwidth measured for the transfer of all *A* and *B* submatrices achieves approximately 1.3 GB/s for this method.

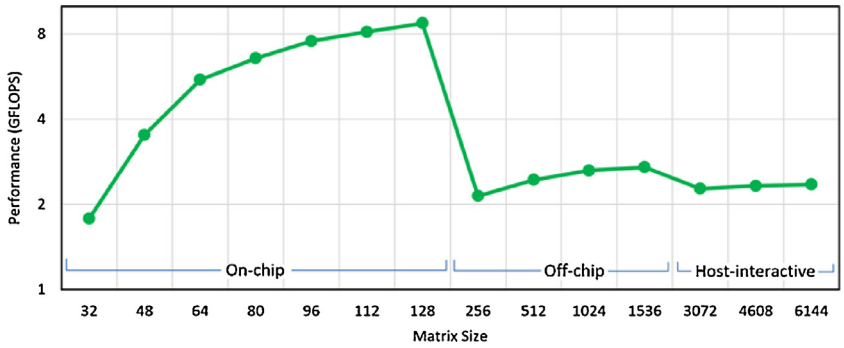

Fig. 5. Performance of the Epiphany III coprocessor for each calculation mode with various problem sizes. Increasing the problem size beyond the available local memory caused a drop in achieved performance due to significant off-chip communication overhead from bandwidth constraints even while using the hardware DMA engine. The limitation of the relatively small 32 MB shared host-coprocessor memory also adds to the software complexity requiring additional host interaction for large problem sizes.

significant implications for software development on this platform. The trade-off in performance for ease of programmability and portability is advantageous and compelling for many application developers. The threaded MPI implementation addresses the critical software development challenge of careful orchestration of data movement in the Epiphany architecture that is, as observed in this study, essential to performance.

6. Related work

The availability of the inexpensive Adapteva Parallella platform with the 16-core Epiphany III processor has led to several investigations of the Epiphany architecture. Most work has relied upon a very primitive programming model using C code for the host and coprocessor interacting through custom communication protocols without the benefit of a parallel programming API. Ul-Abdin et al. [17] implemented autofocus and fast factorized back-projection algorithms on an Epiphany III processor with speedups reported against a sequential CPU design. Varghese et al. [7] evaluated a 5-point stencil kernel and parallel matrix–matrix multiplication using the 64-core Epiphany IV processor, where a comparison between C and hand-tuned assembly code demonstrated the performance attainable with considerable programming effort. Malvoni and Knezovic [18] evaluated Bcrypt hash cracking on Adapteva Parallella and ZedBoard platforms in comparison to CPUs, GPUs, and the Xeon Phi with results showing one to two orders of magnitude improvement in energy efficiency for low-power Epiphany platforms compared with desktop CPUs and GPUs.

Other programming models and APIs for targeting the Epiphany architecture have also been investigated. The COPRTHR SDK [14] provides a direct coprocessor API that includes support for Pthreads [19] semantics as well as support for OpenCL and a thread compiler with Epiphany support. Additional APIs and frameworks that have been investigated include OpenCL within Erlang (based on OpenCL support in the COPRTHR SDK) [20], an APL to C compiler [21], and a CAL compilation framework [22]. Automatically generated code from the dataflow language CAL is compared with a C implementation of the 2D inverse discrete cosine transform targeting the Epiphany architecture in [23]. The development of OpenMP support has also been reported [2]. Significantly, to the best of our knowledge, no investigation has thus far reported on the use of a standard parallel programming API targeting the Epiphany architecture capable of achieving good performance for an algorithm with nontrivial parallelism.

7. Conclusion

We have demonstrated that the Epiphany RISC array architecture can be effectively programmed using conventional MPI syntax. Using a threaded MPI implementation, good performance was observed using nearly unmodified MPI code originally written for a parallel distributed cluster. The proposition of viewing the RISC array as a device-scale parallel platform with a 2D communication topology was validated and led to the design and implementation of an efficient parallel inter-core communication layer within the existing COPRTHR software stack for Epiphany. The availability of a familiar and conventional parallel programming model like MPI should increase productivity for application developers. We have demonstrated that the same MPI code could be extended to treat matrices larger than those that fit in on-chip local memory by copying submatrices from global memory, which becomes the limiting factor in performance. Benchmark results using a conventional MPI implementation for matrix–matrix multiplication compare favorably against the use of hand-tuned assembly and custom inter-core communication code.

References

[1] “Adapteva introduction.” [Online]. Available: <http://www.adapteva.com/introduction/> (accessed 08.01.15).
[2] A. Olofsson, T. Nordström, Z. Ul-Abdin, Kickstarting high-performance energy-efficient manycore architectures with Epiphany, ArXiv Prepr. (2014), ArXiv4125538.
[3] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J.F. Brown III, A. Agarwal, On-chip interconnection architecture of the tile processor, IEEE Micro 27 (September (5)) (2007) 15–31.
[4] M.B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, W. Lee, A. Saraf, N. Shnidman, V. Strumpen, S. Amarasinghe, A. Agarwal, A 16-issue Multiple-Program-Counter Microprocessor with Point-to-Point Scalar Operand Network in 2003, IEEE International Solid-State Circuits Conference (ISSCC) (2003) 170–171.
[5] E16G301 Epiphany 16-core microprocessor, Adapteva Inc. Lexington, MA, Datasheet Rev. 14.03.11.
[6] Parallella-1. x reference manual, Adapteva, Boston Design Solutions, Ant Micro, Rev. 14.09.09.
[7] A. Varghese, B. Edwards, G. Mitra, A.P. Rendell, Programming the Adapteva Epiphany 64-Core Network-on-Chip Coprocessor, in 2014, IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW ’14) (2014) 984–992.
[8] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, Exascale computing study: technology challenges in achieving exascale systems, Def. Adv. Res. vol. 15 (2008).
[9] J. Held, J. Bautista, S. Koehl, From a Few Cores to Many: A Tera-scale Computing Research Overview, Intel Corporation, White Paper, 2006.
[10] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, The Landscape of Parallel Computing Research: A View From Berkeley, University of California, Berkeley, Technical Report UCB/EECS-2006-183, 2006.
[11] S. Borkar, Thousand Core Chips: a Technology Perspective, Proceedings of the 44th Annual Design Automation Conference (DAC ’07) (2007) 746–749.
[12] W.D. Gropp, Learning From the Success of MPI, Proceedings of the 8th International Conference on High Performance Computing (HiPC ’01) (2001) 81–94.
[13] E. Demaine, A Threads-only MPI Implementation for the Development of Parallel Programs, Proceedings of the 11th International Symposium on High Performance Computing Systems (1997) 153–163.

[14] GitHub – The CO-PRocessing THReads (COPRTHR) SDK. [Online]. Available: <https://github.com/browndeer/coprthr> (accessed 08.01.15).
[15] L. Cannon, A cellular computer to implement the kalman filter algorithm, Ph.D. Dissertation Montana State University, 1969.
[16] C. Ozdogan, Cannon's matrix–matrix multiplication with MPI's topologies. [Online]. Available: <http://siber.cankaya.edu.tr/ozdogan/GraduateParallelComputing.old/ceng505/node133html> (accessed 12.01.15).
[17] Z. Ul-Abdin, A. Ahlander, B. Svensson, Energy-Efficient Synthetic-Aperture Radar Processing on a Manycore Architectur, in 2013, 42nd International Conference on Parallel Processing (ICPP '13) (2013) 330–338.
[18] K. Malvoni, J. Knezovic, Are Your Passwords Safe: Energy-Efficient Bcrypt Cracking with Low-Cost Parallel Hardware, 8th USENIX Conference on Offensive Technologies (WOOT '14) (2014).
[19] D. Richie, COPRTHR API reference, Brown Deer Technology, Rev. 1.6.0.0, 2013.
[20] O. Kilic, Explorations in Erlang with the Parallela: a prelude, May-2013. [Online]. Available: <https://www.parallella.org/2013/05/25/explorations-in-erlang-with-the-parallela-a-prelude/> (accessed 08.01.15).
[21] S. Sirlin, The APL C compiler project. [Online]. Available: <http://home.earthlink.net/%E2%88%BCswsirlin/aplcc.html> (accessed 08.01.15).
[22] E. Gebrewahid, M. Yang, G. Cedersjo, Z. Ul-Abdin, V. Gaspes, J.W. Janneck, B. Svensson, Realizing Efficient Execution of Dataflow Actors on Manycores, Proceedings of the 2014 12th IEEE International Conference on Embedded and Ubiquitous Computing (EUC '14) (2014) 321–328.
[23] S. Savas, E. Gebrewahid, Z. Ul-Abdin, T. Nordstrom, M. Yang, An Evaluation of Code Generation of Dataflow Languages on Manycore Architectures, 20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) (2014) 1–9.