Parallel Programming Model for the Epiphany Many-Core Coprocessor Using Threaded MPI

James A. Ross

U.S. Army Research Laboratory
Aberdeen Proving Ground, MD
james.a.ross176.civ@mail.mil

Song J. Park

U.S. Army Research Laboratory
Aberdeen Proving Ground, MD
song.j.park.civ@mail.mil

David A. Richie

Brown Deer Technology
Forest Hill, MD

driche@browndeertechnology.com

Dale R. Shires

U.S. Army Research Laboratory
Aberdeen Proving Ground, MD
dale.r.shires.civ@mail.mil

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.

Categories and Subject Descriptors

C.1.4 [Parallel Architectures]: Miscellaneous;

D.1.3 [Concurrent Programming]: Parallel Programming

General Terms

Benchmarking, Performance, Theory

Keywords

2D RISC Array, MPI, NoC, Many-core, Adapteva Epiphany, Parallella, Energy Efficiency

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. Figure 1 shows the high-level architectural features of the coprocessor. Each of the 16 Epiphany III mesh nodes contains 32 KB 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 mm² on the 65 nm process and 0.128 mm² 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 per watt [Olofsson, personal communication]. The 64-core Epiphany IV, fabricated on the 28 nm process, has demonstrated energy efficiency exceeding 50 GFLOPS per watt [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 1,000 cores in the near future and, shortly thereafter, tens of thousands of cores with an energy efficiency approaching one TFLOPS per watt. 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.

Figure 1. Adapteva Epiphany III architecture diagram

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 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 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 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 subset of pre-existing algorithms implemented with MPI. These calls were sufficient for benchmarking the four algorithms, discussed below, which exhibit various computation and communication patterns and serve as a demonstration of the flexibility of the programming model even with only a partial implementation of the MPI standard.

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 and higher performance over read transactions. The protocol, optimized for the Epiphany architecture, implements a send-receive pair with the receiving thread setting up and enabling the transaction and the sending thread driving the data transfer using a direct write operation to the local memory of the receiving core.

The MPI_Sendrecv_replace call, which provides a mechanism to exchange data between cores, showed remarkable versatility. The call cannot be implemented with a zero-copy design, but rather requires a buffered transaction. The available memory is highly constrained on each Epiphany core, making a large internal MPI buffer extremely costly. Therefore, the implementation must transparently segment large messages into multiple buffered transactions while maintaining high overall performance. This method is similar to the pipelining transport approach implemented in iRCCE [10]. Figure 2 shows the iterative two-stage transfer protocol for sending messages larger than the MPI temporary buffer. The sending thread performs a DMA write to the MPI temporary buffer in the local memory of the receiving thread. When the DMA write transaction has completed, the MPI temporary buffer is then copied into the user application buffer by the receiving thread. These steps are repeated until each segment of the message is transferred. The optimization of this protocol is critical for performance, and only requires synchronization between the sending and receiving thread. Using the benchmarks discussed below, we demonstrate that the protocol does in fact achieve excellent performance in algorithms that rely heavily upon the use of this call. The effect of tuning the size of this small MPI buffer will be examined in detail in section 3.2. MPI tag matching is not implemented and would not be practical due to the small size of the core local memory.

Figure 2. Two-stage synchronous protocol for MPI_Sendrecv_replace transparently segments message into multiple transactions

A threaded MPI implementation has been developed based on the Epiphany support provided by the COPRTHR SDK [11]. 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. Threaded MPI exhibits the highest performance reported to date using a standard parallel API for the Epiphany architecture.

Table 1. List of MPI calls used in four applications

MPI Library Calls	
MPI Init	MPI Cart coords
MPI Finalize	MPI Cart shift
MPI Comm size	MPI Sendrecv replace
MPI Comm rank	MPI Send
MPI Cart create	MPI Recv
MPI Comm free	

3. APPLICATIONS

Our previous demonstration of dense matrix-matrix multiplication using threaded MPI [5] has been optimized and extended to include four algorithms commonly used for benchmarking: matrix-matrix multiplication, N-body particle interactions, five-point 2D stencil update, and 2D FFT. The work in [5] included an early prototype of the threaded MPI implementation and a single matrix-matrix multiplication application. The threaded MPI library communication performance and overhead, in addition to the computational portion of the linear algebra operation have been optimized for performance. Each of these applications in this paper exhibit different computational and communication scaling, as shown in Table 2. Scaling is shown with respect to either the side-length n of the problem size or in the case of N-body particle interaction, the total size N for the problem. The performance numbers reported in this paper are for on-chip time only, excluding the startup cost to initialize the cores and copy data from off-chip memory. Reference code for the threaded MPI applications discussed in this paper are available on GitHub [12]. An example of application code using the threaded MPI programming model with MPI_Sendrecv_replace appears in Listing 1.

Table 2. Computation and communication algorithm scaling

Application	Computation	Communication
Matrix-matrix multiply	$O(n^3)$	$O(n^2)$
N-body [†]	$O(N^2)$	$O(N)$
5-point 2D stencil [†]	$O(n)$	$O(n)$
2D FFT	$O(n^2 \cdot \log_2(n))$	$O(n^2)$

```

1 void my_thread( void* p ) {
2   my_args_t* args = (my_args_t*)p;
3   int n = args->n; // submatrix dimension
4   int rank2d, coords[2];
5   int dim[2] = {args->d, args->d};
6   int per[2] = {1, 1}; // periodic network
7   MPI_Status s;
8   MPI_Comm com2d, com = MPI_COMM_THREAD;
9
10  // Initialize MPI parameters
11  MPI_Init(0, MPI_BUF_SIZE);
12  MPI_Cart_create(com, 2, dim, per, 1, &com2);
13  MPI_Comm_rank(com2d, &rank2d);
14  MPI_Cart_coords(com2d, rank2d, 2, coords);
15
16  // Compute ranks of the up and left shifts
17  int up, down, left, right;
18  MPI_Cart_shift(com2d, 0, 1, &left, &right);
19  MPI_Cart_shift(com2d, 1, 1, &up, &down);
20
21  // Allocate buffs, init. w/ DMA global mem.
22  ...
23  for (int i = 1; i < args->d; i++) {
24    MatrixMultiply(n, a, b, c);
25    // Shift matrix a left, shift matrix b up
26    MPI_Sendrecv_replace(a, n*n, MPI_FLOAT,
27      left, 1, right, 1, com2d, &s);
28    MPI_Sendrecv_replace(b, n*n, MPI_FLOAT,
29      up, 1, down, 1, com2d, &s);
30  }
31  MatrixMultiply(n, a, b, c);
32  // Write c result buffer to global memory
33  ...
34  MPI_Finalize();
35 }

```

Listing 1. Example code for matrix multiplication application

The internal MPI buffer size used for buffered transfer protocols like that employed by the `MPI_Sendrecv_replace` routine is tunable and was selected to achieve the highest performance for each application within the limitations of the 32 KB local memory per core. The specific buffer size selected was 1.5 KB used for matrix-matrix multiplication, 1 KB for N-body simulation, 256 bytes for the 5-point stencil calculation, and 512 bytes for the 2D FFT. There is no algorithm for determining how much data to use for the internal MPI buffer although it needn't be larger than the largest MPI buffer to be sent. The marginal performance improvement for a larger internal MPI buffer decreases and also may decrease the largest problem size that can fit within the core local memory.

Each application exhibits different on-chip communication patterns as shown in Figure 3. The matrix-matrix multiply application uses a 2D periodic domain decomposition where each A submatrix is rotated left through each row and the B submatrix is rotated up through each column. The N-body application uses a

1D periodic domain decomposition where communication moves in a scan line manner. Each core in the five-point 2D stencil application communicates with nearest neighbors in a 2D domain. The 2D FFT uses a 1D periodic domain and all-to-all communication.

Figure 3. Application inter-core communication patterns

3.1 Roofline Model

In order to characterize the expected performance of each of the applications explored, we apply a roofline model [13] to the architecture and evaluate the arithmetic intensity (AI) of the computational kernel of each application. Table 3 shows the largest problem sizes, the number of floating point operations, the amount of data used, and the calculated AI for each application. The largest problem sizes are determined by the core local memory available after the program, internal MPI buffer, and stack are factored in. In some cases, larger problems can be executed but are either not convenient powers of two or require decreasing the MPI buffer size and sacrificing communication performance.

Table 3. Arithmetic intensity for largest problem sizes

Application	Max n/N	MFLOPS	KBytes	AI
Matrix multiply	128	4.19	192	21.33
N-body [†]	4096	335	192	2048
5-pt 2D stencil [†]	144	0.19	162	1.13
2D FFT	128	1.15	128	8.75

Each Epiphany III core can perform a double word (64-bit) load or store each cycle, although analysis of the machine code generated for these applications uses single word (32-bit) loads. It also has instructions for a fused multiply-add (FMA) and fused multiply-subtract, enabling applications to achieve up to two floating point operations per cycle if the arithmetic has a balanced number of multiply and addition (or subtraction) operations. Access to core local memory is significantly faster than off-chip global DRAM. The Parallella platform uses custom FPGA logic to control off-chip access to global DRAM and can be improved in the future. As of June, 2015, the measured read and write performance is about 140 MB/s on a single DMA channel. Figure

[†] Both N-body and 5-point 2D stencil methods are iterative, but the analysis uses only a single iterations of the algorithms

4 shows the attainable single precision performance from the Epiphany III platform. For applications that have a low arithmetic intensity or access global DRAM, the attainable performance is low. Empirical measurements place global DRAM writes around 236 MB/s and reads around 86 MB/s. The geometric mean of about 140 MB/s for the global DRAM is used to characterize the roofline model for this paper. The off-chip bandwidth in the currently available hardware presents practical limits to attainable host-interactive performance for many applications, but it is expected that future iterations of the architecture will improve upon this.

Figure 4. Epiphany III roofline model for 16 cores, 600 MHz

The roofline model presents us with upper limits to the expected performance for applications when we understand the problem size, memory access, and floating point operation ratios. It also suggests that as the number of cores increases, even without a proportional scaling in off-chip bandwidth, the architecture may have improved computational efficiency. This is the case for classes of algorithms where the arithmetic intensity increases with the problem size (the computational scaling is greater than the communication scaling) as is the case for all applications explored here with the exception of the 5-point 2D stencil update application. The roofline model does not account for the costs of necessary communication for non-trivial parallel applications. To fully understand the application performance, an inter-core communication latency and bandwidth model is required.

3.2 MPI Latency and Bandwidth

The `MPI_Sendrecv_replace` routine was used for inter-core communication in each of the four applications. The routine was critical to achieving high communication performance. Figure 5 shows the effective inter-core bandwidth of the routine for a 2D periodic network topology where every core transfers data to the west and receives data from the east. This regular communication pattern achieves the highest total on-chip bandwidth, whereas a random communication pattern would be limited by the bisection bandwidth. The performance is a function of the total amount of data transferred and the internal MPI buffer size. As expected, the effective bandwidth improves with increasing internal MPI buffer size, with peak inter-core bandwidth performance for typical application transfers approaching 1,000 MB/s, which is equivalent to 80% of the theoretical peak of about 1,250 MB/s for a single-channel DMA transfer for the Epiphany III.

The data was fit to a standard alpha-beta model for MPI communication modified to account for the latency dependence on the internal MPI buffer size. Specifically, the latency will increase with the number of internal DMA transactions used to

communicate a message of a given size. As the internal MPI buffer size is increased, the number of required transactions will decrease proportionately. Taking this into account, the total communication time T is modeled as $T = \alpha_0 + \alpha_1 \cdot k + \beta \cdot m$, where α_0 is the fixed latency of the MPI call, α_1 is the latency per buffered DMA transaction, and β is the bandwidth. Here, m is the message size in bytes and k is the number of internal DMA transactions, calculated as m divided by the MPI buffer size. The measured data fit to determine the values for the latency and bandwidth parameters and found to be $\alpha_0 = 1,216$ ns, $\alpha_1 = 309$ ns, $\beta = 1,250$ MB/s.

Figure 5. Effective inter-core bandwidth of the `MPI_Sendrecv_replace` call for different MPI buffer sizes

Significantly, the Epiphany architecture bandwidth scales with the number of on-chip cores. This differs from many other CPU architectures and the Xeon Phi where the effective inter-core bandwidth for an MPI code is limited by the bandwidth of the ring bus and may have higher clock latency due to the complex cache hierarchy.

Future improvements in the threaded MPI implementation could include a crossover point that replaces the use of the DMA engine in favor of direct inter-core write operations for small message sizes. This would have the greatest effect on performance of grid applications where small messages of only boundary data are shared between neighboring cores.

3.3 Matrix–Matrix Multiplication

The dense matrix–matrix multiplication MPI code based on Cannon’s algorithm for square matrices was previously developed for a conventional parallel cluster [14] and adapted here with minimal modification. In the context of the Epiphany coprocessor, the first communication step, which skewed the submatrices data across the cores, was removed as it is unnecessary when using a temporary copy of the host matrices. Instead, the submatrices were read in from main memory preskewed and the B submatrix was transposed for a more efficient memory access pattern within the inner loop of the matrix–matrix multiply. The final communication step to reorder the data was also removed since it was unnecessary because it was an intermediate copy of the host data. The three inner loops of the matrix–matrix multiplication were unrolled by four and a fused multiply–add GCC built-in function was used to force the compiler to generate the most efficient code. The inner loop then

demonstrated operation at the peak performance of the core. This by itself is significant insofar as the compiler is observed to generate highly optimized instructions from C code.

The reported benchmarks used an internal MPI buffer size of 1.5 KB, although little performance improvement is seen beyond 512 bytes. The performance of the matrix–matrix multiplication code in GFLOPS is calculated using the formula $2 \cdot n^3 / t$, where n , is the matrix size in a single dimension and t is the total execution time in nanoseconds. Figure 6 shows the overall measured performance of the full matrix–matrix calculation on 16 cores including the blocking communication time between the cores.

Figure 6. On-chip matrix–matrix multiplication performance

For each workload, the *relative time* spent for computation and communication is estimated and shown visually as a relative fraction of the represented performance. This algorithm exhibits a roughly even breakdown between computation and communication costs, with slightly more time spent on computation at larger workloads. Despite the communication taking a significant fraction of the execution time, the application is able to achieve 12.02 GFLOPS. This performance result exceeds 20 GFLOPS per watt. This is about 63% of the peak performance of the processor and 74% (12.02 GFLOPS vs. 16.2 GFLOPS) of the performance achieved in [15] for a 4×4 array of cores on the Epiphany IV processor, where low-level optimized assembly and custom communication calls with a double-buffer scheme were used. It is also faster than our initial published result of 9.1 GFLOPS [5] because of the improved communication bandwidth, reduced overhead, and optimized matrix-matrix multiply routine.

For comparison, an OpenCL implementation of matrix-matrix multiplication was developed using the threaded MPI base code. The OpenCL kernel lacked any extensions for inter-core communication or off-chip DMAs but did read submatrix data from global memory into core local “private” memory before computation. The implementation obtained a peak performance result of 16 MFLOPS for the 128x128 case. This result is significantly inferior to the threaded MPI result where on-chip memory reuse and inter-core communication are required to achieve high performance on the architecture. OpenCL without necessary extensions for the Epiphany architecture is not an appropriate programming model for applications with a low to moderate arithmetic intensity. Fundamentally, the logical OpenCL device architecture accounts for memory locality in but neglects explicit memory access, instead favoring implicit hierarchical memory access. Device architectures built to access OpenCL private or local memory from neighboring compute units

must introduce non-standard API extensions. Thus, the portability argument for OpenCL is lost and the Epiphany programmer must make a subjective determination whether to use a standard interface for explicit control of inter-core data movement. Both host APIs for the COPRTHR threaded MPI and COPRTHR STDCL interface to OpenCL are very similar. It took less than one hour to port both host and device codes from threaded MPI to OpenCL.

Discounting communication time, the computational core of the application achieves over 99% of peak performance within the context of the roofline model of the architecture. This was achieved by unrolling the two innermost loops by four and explicitly using the built-in FMA intrinsic function. This utilized a high number of the 64 general-purpose registers. The compiler was then able to generate efficient machine code so that the Epiphany dual-issue scheduler could execute concurrent load and FMA instructions in parallel. Improvements to the application performance can only be made by addressing the inter-core communication with non-blocking communication schemes to overlap computation with communication. The arithmetic intensity of the application (see Table 3) is high enough that one can consider using the Epiphany cores as a coprocessor to the host by streaming data from global DRAM. In this scenario, the peak measured performance from global DRAM access was 2.7 GFLOPS. This is in close agreement with the roofline model for the architecture shown in Figure 4.

Larger problems that do not completely fit within 32 KB per core local memory may be executed by streaming data from global memory. The code was augmented to enable streaming off-chip access so that partial matrix multiplications are solved incrementally in 128x128 submatrix blocks. The total performance in this execution mode was then limited by off-chip global memory bandwidth, peaking in performance around 2.7 GFLOPS at a matrix size of 1536x1536. This poor performance result highlights the off-chip bandwidth limitations of the currently available hardware. It approaches the expected peak performance attainable with the roofline model of the architecture for this application with an arithmetic intensity of about 21.

3.4 N-body Particle Interaction

The N-body code was based on an existing MPI implementation of the algorithm [16]. The code was extended from two to three dimensions; the MPI_Isend and MPI_Irecv pair was replaced with MPI_Sendrecv_replace; and then the code was optimized for performance. This algorithm has the most favorable compute-to-communication ratio of the four algorithms examined in this work; the application was generally not communication-bound and can be considered a best-case scenario for nontrivial parallelism. It is also iterative so that once the particle data is on-chip, there is a significant amount of computation. Unlike the other applications, the N-body calculation uses a 1D communication topology so that data moved in a single scan line cycle. A network topology based on a continuous fractal space-filling curve was explored but had a negligible effect on the overall performance. High performance was achieved by unrolling the outer compute loop by eight, explicitly using the FMA built-in function, and using a fast approximation for the inverse square root. For each particle position update, the cores shift the working set of particle positions and masses to the neighboring core. Once the working set has completely cycled

through all the cores, the working set is updated with the new particle positions and the next time step iteration takes place.

The performance in GFLOPS is calculated using the convention $20 \cdot i \cdot N^2 / t$ where i is the number of time step iterations, N is the number of particles, and t is the total execution time in nanoseconds. The convention considers the inverse square root to be two floating point operations although it takes multiple cycles to compute. The use of a software square root operation and deviation from a one-to-one ratio of multiply/add operations make it impossible to achieve the theoretical peak for this particular algorithm. Benchmarks reported used an internal MPI buffer of 1,024 bytes. As shown in Figure 7, the *relative time* spent on communication was significantly low so that the internal buffer could have been much smaller. Any internal MPI buffer of more than 64 bytes would have been sufficient to achieve negligible relative communication time beyond 1,024 particles. The highest measured performance of 8.28 GFLOPS is 43% of peak performance, but the full analysis isn't as simple as the matrix-matrix multiply application.

Figure 7. On-chip N-body particle interaction performance

Considering the software inverse square root, there are actually 29 floating point and integer operations from 22 instructions for each particle update in the inner loop. Operational performance is measured to be 12.05 GOPS from a theoretical 12.65 GOPS peak, so this application achieves over 95% of expected performance for this architecture. The on-chip work size is limited to fewer than approximately 5,000 particles. Because of the exceptionally high arithmetic intensity of this algorithm, shown in Table 3, larger calculations can be performed by streaming data from global memory and throughput only marginally decreases (less than 5%).

3.5 Five-Point 2D Stencil

The 5-point 2D stencil update was implemented using a conventional parallelization with MPI. Other researchers have used this grid application to evaluate performance on the Epiphany [15] and similar architectures [17]. This analog for many finite difference approximations combines the grid point itself along with neighboring data in the cardinal directions, scales these values by a constant, and stores the result for subsequent iterations. Of all of the applications, this algorithm has the least favorable computation-to-communication ratio so that performance is mostly limited by communication. However, the algorithm is iterative so there is a significant amount of computation once the data is on-chip. For each point, there is one multiplication and four FMA operations using five values loaded from memory. Implementation of the benchmark used

conventional parallelization with MPI on a 2D network topology. Communication occurs among each of the four cardinal directions per iteration. The 2D computational domain is distributed across all cores such that it mirrors the physical network layout. Network domain and physical domain boundaries are not shared and the data values are kept fixed. Each communication buffer is packed into a temporary storage buffer before sending and receiving. The inner two loops of the algorithm, iterating by rows and then by columns, were optimized by unrolling each by four so that a 4x4 block of data and edges were loaded into registers for optimal data re-use. Additionally, after each column block shift right, the previous edge values were reused for the next block calculation. The performance in GFLOPS is calculated using the formula $9 \cdot i \cdot n^2 / t$ where i is the number of iterations, n is the problem size along one dimension, and t is the execution time in nanoseconds.

Figure 8. On-chip 5-point 2D stencil performance

The reported benchmarks used an internal MPI buffer of 256 bytes. For each workload, the *relative time* spent for computation and communication is estimated and shown visually as a relative fraction of the represented performance in Figure 8. The highest measured performance of 6.35 GFLOPS is 33% of peak performance. Part of the reason for the low performance is that the effective inter-core bandwidth is low due to the small buffer sizes. In the case of the 128x128 workload, each edge buffer is 128 bytes. As shown in Figure 5, one should not expect very high communication performance but rather less than 100 MB/s. Since the transfer sizes are small, it makes sense to explore other MPI communication mechanisms where multiple transfers can be queued in order to reduce the amount of global synchronization. For example, the one-sided communication routine `MPI_Put` makes the most sense for this application where a core can write small buffers directly to neighboring core memory. For small buffers such as those in this application, direct writes have higher effective bandwidth than using the DMA engine.

Considering just the computational kernel of the application, the code has a roofline performance limit of 17.28 GFLOPS due to a floating point operation imbalance of four fused multiply-add operations and one multiply operation. The computation portion measured over 13.2 GFLOPS or 76.4% of our expected roofline limit. So there is some room left for optimization on the computational core, but it may be challenging because the arithmetic intensity (Table 3) and register reuse are relatively low.

3.6 2D Fast Fourier Transform

The complex 2D FFT application is parallelized over stripes and performs a 1D radix-2 decimation-in-time (DIT) FFT, a corner

turn, an additional radix-2 DIT FFT, and a final corner turn using the Cooley–Tukey algorithm. The implementation of the benchmark used a conventional parallelization with MPI. At small workloads, the performance is dominated by the corner turn communication. For improved performance the inner loop of the in-place radix-2 DIT code was unrolled by two. This unroll is unlike the other applications and nontrivial. Additionally, the complex data type is less amenable to optimization with FMA built-in functions. The performance in GFLOPS is calculated using the standard formula for complex 2D FFT used by the FFTW group of $5 \cdot n^2 \cdot \log_2(n^2)/t$ where n is the problem size along one dimension and t is the execution time in nanoseconds (note that this is not a precise FLOP count but, rather, a convenient scaling for comparison to other algorithms) [18].

The reported benchmarks used an internal MPI buffer of 512 bytes. For each workload, the *relative time* spent for computation and communication is estimated and shown visually as a relative fraction of the represented performance in Figure 1. The highest measured performance of 2.50 GFLOPS is 13% of peak performance. Despite exhibiting the lowest percent of peak performance of the four algorithms examined, the result is quite good in comparison to other CPU architectures, particularly on using a power-efficiency metric.

Figure 9. On-chip 2D fast Fourier transform performance

Considering a roofline model of the computational core, the inner loop of the radix-2 butterfly operation contains a complex multiplication, a complex addition, and a complex subtraction. This compiles to one fused multiply–add, one fused multiply–subtraction, two floating point multiplications, two floating point additions, and two floating point subtractions for a roofline performance limit of 12 GFLOPS. Discounting communication time, the computational core achieves 4.2 GFLOPS or 35% of the expected roofline performance limit. This is lower than expected and an analysis of the machine code shows that the compiler did not use double-word loads and stores for accessing the 64-bit complex data. That may account for about 36% of the expected performance while the remaining performance difference is from array index calculations for the complex butterfly memory access pattern. The 2D FFT is a challenging application to optimize in both computation and communication portions.

4. DISCUSSION

The benchmarks for on-chip performance demonstrate that threaded MPI is generally applicable for a number of applications on the Epiphany architecture. The benchmark trends suggest that the performance would improve if the Epiphany cores had more than 32 KB of available shared local memory, and the Epiphany road map indicates that this change will occur in future

generations. Typically, the program and stack use approximately 16 KB of the available memory, so doubling the shared local memory to 64 KB would effectively triple the usable application memory to about 48 KB. This generally has the effect of increasing the amount of local work per core, increasing the arithmetic intensity for some applications, reducing the relative amount of communication, and improving overall performance and performance per core. Alternatively, larger programs could be executed with more core memory.

The lack of a hardware cache at any level on the Epiphany architecture has many benefits, but at the expense of requiring good software design. Direct access to the local shared memory is very fast, roughly equivalent to the speed of L1 cache. Removing the cache enables a higher core count within the same die area or lower power consumption. These design decisions directly impact the ability to scale to thousands of cores in the near future.

The demonstrated performance on Epiphany, as a percentage of peak performance, compares very favorably with other coprocessors such as the Intel Xeon Phi and Intel Teraflops Research Chip. The Intel Xeon Phi 5110P achieves 86% of peak performance on single precision general matrix multiply (SGEMM) using the Intel Math Kernel Library (MKL) [19], although achievable performance using a standard parallel API such as OpenMP or MPI is typically much lower. Vangal et al. [17] reported the 80-core Intel Teraflops Research Chip achieved 37.5% of peak performance on matrix multiplication, 2.73% of peak performance on 2D FFT using the same algorithm, and 73.3% of peak performance on a 2D stencil code with a different algorithm and communication pattern.

5. RELATED WORK

Adapteva’s Epiphany SDK includes ANSI C/C++ support [20]. Other methods of programming include COPRTHR SDK [11], Erlang [21], Array Manipulation Language [22], and CAL Actor Language with Network Language [23]. A model based on message passing paradigm is not known to have been implemented.

In terms of applications, a few studies have surveyed programming and performance of the Epiphany architecture. Matrix multiplication was tested in Sapir’s work [24]. The 2D FFT was benchmarked in Sapir’s other work [25]. The back-projection algorithm was implemented on Epiphany in Zain-Ul-Abdin et al. [26]. Stencil and matrix multiplication was analyzed for the 64-core Epiphany in Varghese et al. [15]. Hash decryption on Parallella board was tested in Malvoni and Knezovic’s work [27]. However, these previous studies used the Epiphany SDK directly and without a standard programming model and API.

As for hardware, several prior works exist in tiled many-core architectures. Intel has developed tile-based, many-core prototypes Teraflop Research Chip [17] and Single-chip Cloud Computer (SCC) [28]. Tilera has produced a tiled cache-coherent TILE64 [29] inspired by MIT’s RAW processor [30]. Ambric introduced RISC with a massively parallel processor array chip Am2045 [31]. Scorpio [32] is an academic 36-core chip design at the Massachusetts Institute of Technology. STMicroelectronics has designed a many-core processor Platform 2012 [33], also known as STHORM [34]. Kalray has released a high-performance, low-power MPPA-256 chip [35]. Typically, these architectures have low-level primitives for inter-core

communication, some basic communication libraries, and some MPI implementations at various feature levels.

Thread support for MPI on shared memory systems has been examined in the past. Protopopov and Skjellum proposed a multithreaded MPI design in [36]. Demaine presented a threads implementation of MPI where a single process and threads are used for each MPI process in [37]. Similarly, Tang et al. studied thread-based MPI system which maps each MPI node to a thread [38].

Although SCC consumes an order magnitude greater power consumption, a tiled IA cores resemble the Epiphany architectural layout. Message passing API RCCE [39] and lightweight MPI implementation RCKMPI [40] were developed for the SCC processor. Similar to our work, the many-core processor was programmed with MPI codes. Mattson et al. [10] argue that the SCC processor is a message passing chip at its core and thereby the most efficient programming model should have the ability to send messages between cores. However, there is a notable distinction of the Epiphany chip being represented as a coprocessor in currently available platforms.

6. CONCLUSIONS

We demonstrated, using a standard API, that threaded MPI provides a programming model capable of achieving high performance and efficiency for a range of parallel applications on the Epiphany many-core architecture without employing hand-tuned assembly or custom inter-core communication code. The proposition of viewing the architecture as a device-scale parallel networked cluster with a 2D communication topology was further validated. We demonstrated that previously written MPI software can be ported to the Epiphany with little modification, simplifying the processes of domain decomposition and application development. Benchmark results using a conventional MPI implementation compare favorably with the matrix-matrix multiplication, 2D FFT, and 2D stencil update performance on similar architectures.

7. FUTURE WORK

Future work will investigate the application of non-blocking zero-copy and one-sided MPI communication calls to enable an overlap of computation and communication. This is expected to increase performance for the matrix-matrix multiplication and stencil applications. A more complete network performance model for MPI, only briefly discussed in section 3.2, will be developed and compared to other modern architectures such as the Intel Xeon Phi. Image processing methods and artificial neural networks will be explored as well as additional applications common in high performance computing such as the conjugate gradient method.

8. ACKNOWLEDGMENTS

The authors wish to acknowledge the U.S. Army Research Laboratory-hosted Department of Defense Supercomputing Resource Center for its support of this work.

9. REFERENCES

- [1] "Epiphany architecture reference," Adapteva, Rev. 14.03.11.
- [2] "E16G301 Epiphany 16-core microprocessor," Adapteva Inc., Lexington, MA, Datasheet Rev. 14.03.11.
- [3] "Parallella-1.x reference manual," Adapteva, Boston Design Solutions, Ant Micro, Rev. 14.09.09.
- [4] A. Olofsson, T. Nordström, and Z. Ul-Abdin, "Kickstarting high-performance energy-efficient manycore architectures with Epiphany," *ArXiv Prepr. ArXiv14125538*, 2014.
- [5] D. Richie, J. Ross, S. Park, and D. Shires, "Threaded MPI programming model for the Epiphany RISC array processor," in *Accepted for publication in Special Issue of the Elsevier Journal of Computational Science*, 2015.
- [6] J. A. Ross, D. A. Richie, S. J. Park, and D. R. Shires, "Parallel Programming Model for the Epiphany Many-Core Coprocessor Using Threaded MPI," in *Proceedings of the 3rd International Workshop on Many-core Embedded Systems (MES'15)*, 2015, pp. 41–47.
- [7] "MPI 'lite' proof-of-concept," *Parallella Community*. [Online]. Available: <https://parallella.org/forums/viewtopic.php?f=17&t=95>. [Accessed: 26-Mar-2015].
- [8] W. Gropp and R. Thakur, "Thread-safety in an MPI implementation: Requirements and analysis," *Parallel Comput.*, vol. 33, no. 9, pp. 595–604, Sep. 2007.
- [9] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "Toward efficient support for multithreaded MPI communication," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Springer, 2008, pp. 120–129.
- [10] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and improvements of programming models for the Intel SCC many-core processor," in *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, 2011, pp. 525–532.
- [11] D. Richie, "COPRTHR API reference," Brown Deer Technology, Rev. 1.6.0.0, 2013.
- [12] "Threaded MPI examples for the Adapteva Epiphany architecture," *GitHub*. [Online]. Available: <https://github.com/USArmyResearchLab/mpi-epiphany>. [Accessed: 30-Jun-2015].
- [13] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.
- [14] "Matrix-Matrix Multiply." [Online]. Available: https://svn.mcs.anl.gov/repos/performance/benchmarks/mpi_overlap/matmul.c. [Accessed: 23-Mar-2015].
- [15] A. Varghese, B. Edwards, G. Mitra, and 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, pp. 984–992.
- [16] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk, "N-body program using pipelining algorithm." [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples-usingmpi/advmsg/nbodypipe_c.html. [Accessed: 23-Mar-2015].
- [17] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar, "An 80-tile sub-100-W teraFLOPS processor in 65-nm CMOS," *IEEE J. Solid-State Circuits*, vol. 43, no. 1, pp. 29–41, Jan. 2008.
- [18] M. Frigo and S. Johnson, "FFT Benchmark Methodology," *benchFFT*. [Online]. Available:

- <http://www.fftw.org/speed/method.html>. [Accessed: 26-Jun-2015].
- [19] “Intel® Xeon Phi™ Coprocessor SGEMM* and DGEMM* Throughput,” *Intel*. [Online]. Available: <http://www.intel.com/content/www/us/en/benchmarks/server/xeon-phi/xeon-phi-sgemm-dgemm.html>. [Accessed: 29-Oct-2015].
- [20] “Epiphany SDK Reference,” Adapteva Inc.
- [21] “Erlang-OTP and the Parallella Board,” Mar-2015. [Online]. Available: <https://www.parallella.org/2015/03/20/erlang-otp-and-the-parallella-board/>. [Accessed: 25-Mar-2015].
- [22] B. Webber, “The Apl to C compiler aplc is now ported to the Parallella.” [Online]. Available: <http://forums.parallella.org/viewtopic.php?f=13&t=338#p2016>. [Accessed: 25-Mar-2015].
- [23] E. Gebrewahid, M. Yang, G. Cedersjo, Z. Ul-Abdin, V. Gaspes, J. W. Janneck, and B. Svensson, “Realizing efficient execution of dataflow actors on manycores,” in *Proceedings of the 2014 12th IEEE International Conference on Embedded and Ubiquitous Computing (EUC '14)*, 2014, pp. 321–328.
- [24] Y. Sapir, “Scalable parallel multiplication of big matrices,” Adapteva, Inc., White Paper, 2012.
- [25] Y. Sapir, “Using a scalable parallel 2D FFT for image enhancement,” Adapteva, Inc., White Paper, 2012.
- [26] Z. Ul-Abdin, A. Ahlander, and B. Svensson, “Energy-efficient synthetic-aperture radar processing on a manycore architecture,” in *2013 42nd International Conference on Parallel Processing (ICPP '13)*, 2013, pp. 330–338.
- [27] K. Malvoni and J. Knezovic, “Are your passwords safe: energy-efficient bcrypt cracking with low-cost parallel hardware,” in *8th USENIX conference on Offensive Technologies (WOOT'14)*, 2014.
- [28] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, and others, “A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS,” in *International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2010, pp. 108–109.
- [29] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, “On-chip interconnection architecture of the tile processor,” *IEEE Micro*, vol. 27, no. 5, pp. 15–31, Sep. 2007.
- [30] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, and others, “Evaluation of the RAW microprocessor: An exposed-wire-delay architecture for ILP and streams,” *31st Annu. Int. Symp. Comput. Archit. ISCA 04*, pp. 2–13, 2004.
- [31] M. Butts, A. M. Jones, and P. Wasson, “A structural object programming model, architecture, chip and tools for reconfigurable computing,” in *Field-Programmable Custom Computing Machines (FCCM)*, 2007, pp. 55–64.
- [32] B. K. Daya, C.-H. O. Chen, S. Subramanian, W.-C. Kwon, S. Park, T. Krishna, J. Holt, A. P. Chandrakasan, and L.-S. Peh, “SCORPIO: A 36-core research chip demonstrating snoopy coherence on a scalable mesh NoC with in-network ordering,” in *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA'14)*, 2014, pp. 25–36.
- [33] D. Melpignano, L. Benini, E. Flamand, B. Jogo, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit, “Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications,” in *49th Annual Design Automation Conference (DAC '12)*, 2012, pp. 1137–1142.
- [34] A. Marongiu, A. Capotondi, G. Tagliavini, and L. Benini, “Improving the programmability of STHORM-based heterogeneous systems with offload-enabled OpenMP,” in *First International Workshop on Many-core Embedded Systems (MES'13)*, 2013, pp. 1–8.
- [35] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel, “A distributed runtime environment for the Kalray MPPA®-256 integrated manycore processor,” *Procedia Comput. Sci.*, vol. 18, pp. 1654–1663, 2013.
- [36] B. Protopopov and A. Skjellum, “A Multithreaded Message Passing Interface (MPI) Architecture: Performance and Program Issues,” *J. Parallel Distrib. Comput.*, vol. 61, no. 4, pp. 449–466, Apr. 2001.
- [37] E. D. Demaine, “A threads-only MPI implementation for the development of parallel programs,” in *Proceedings of the 11th International Symposium on High Performance Computing Systems*, 1997, pp. 153–163.
- [38] K. Shen, H. Tang, and T. Yang, “Adaptive two-level thread management for fast MPI execution on shared memory machines,” in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, 1999, p. 49.
- [39] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and others, “The 48-core SCC processor: the programmer’s view,” in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*, 2010, pp. 1–11.
- [40] I. A. C. Ureña, M. Riepen, and M. Konow, “RCKMPI—lightweight MPI implementation for Intel’s Single-chip Cloud Computer (SCC),” in *18th European MPI Users’ Group conference on Recent advances in the message passing interface (EuroMPI'11)*, 2011, pp. 208–217.