Remote Procedure Calls for Improved Data Locality with the Epiphany Architecture

James A. Ross[1(✉)] and David A. Richie[2]

[1] U.S. Army Research Laboratory, Aberdeen Proving Ground, MD 21005, USA
james.a.ross176.civ@mail.mil

[2] Brown Deer Technology, Forest Hill, MD 21050, USA
drichie@browndeertechnology.com

Abstract. This paper describes the software implementation of an emerging parallel programming model for partitioned global address space (PGAS) architectures. Applications with irregular memory access to distributed memory do not perform well on conventional symmetric multiprocessing (SMP) architectures with hierarchical caches. Such applications tend to scale with the number of memory interfaces and corresponding memory access latency. Using a remote procedure call (RPC) technique, these applications may see reduced latency and higher throughput compared to remote memory access or explicit message passing. The software implementation of a remote procedure call method detailed in the paper is designed for the low-power Adapteva Epiphany architecture.

Keywords: Remote procedure call (RPC) · Network-on-chip (NoC) Distributed computing · Partitioned global address space (PGAS) Programming model

1 Introduction and Motivation

Many high performance computing (HPC) applications often rely on computer architectures optimized for dense linear algebra, large contiguous datasets, and regular memory access patterns. Architectures based on a partitioned global address space (PGAS) are enabled by a higher degree of memory locality than conventional symmetric multiprocessing (SMP) with hierarchical caches and unified memory access. SMP architectures excel at algorithms with regular, contiguous memory access patterns and a high degree of data re-use; however, many applications are not like this. A certain class of applications may express irregular memory access patterns, are "data intensive" (bandwidth-heavy), or express "weak locality" where relatively small blocks of memory are associated. For these applications, memory latency and bandwidth drive application performance. These applications may benefit from PGAS architectures and the re-emerging remote procedure call (RPC) concept. By exporting the execution of a program to the core closely associated with the data, an application may reduce the total memory latency and network congestion associated with what would be remote direct memory access (RDMA).

2 Background and Related Work

The 16-core Epiphany-III coprocessor is included within the inexpensive ARM-based single-board computer "Parallella" [1]. All of the software tools and firmware are open source, enabling rigorous study of the processor architecture and the exploration of new programming models. Although not discussed in detail in this paper, the CO-PRocessing Threads (COPRTHR) 2.0 SDK [2] further simplifies the execution model to the point where the host code is significantly simplified, supplemental, and even not required depending on the use case. We also use the *ARL OpenSHMEM for Epiphany* library in this work [3]. Currently, a full implementation of OpenSHMEM 1.4 is available under a BSD open source license [4]. The combination of the COPRTHR SDK and the OpenSHMEM library enabled further exploration of hybrid programming models [5], high-level C++ templated metaprogramming techniques for distributed shared memory systems [6].

The middleware and library designs for Epiphany emphasize a reduced memory footprint, high performance, and simplicity, which are often competing goals. The OpenSHMEM communications library is designed for computer platforms using PGAS programming models [7]. Historically, these were large Cray supercomputers and then commodity clusters. But now the Adapteva Epiphany architecture represents a divergence in computer architectures typically used with OpenSHMEM. In some ways, the architecture is much more capable than the library can expose. The architecture presents a challenge in identifying the most effective and optimal programming models to exploit it. While OpenSHMEM does reasonably well at exposing the capability of the Epiphany cores, the library does not provide any mechanism for RPCs.

Recent publications on new computer architectures and programming models have rekindled an interest in the RPC concept to improve performance on PGAS architectures with non-uniform memory access to non-local memory spaces. In particular, the Emu System Architecture is a newly developed scalable PGAS architecture that uses hardware-accelerated "migrating threads" to offload execution to the remote processor with local access to application memory [8]. The Emu programming model is based on a partial implementation of the C language extension, Cilk. Cilk abandons C semantics and the partial implementation is used for little or no improvement in code quality over a simple C API. The Emu software and hardware RPC implementation details are not publicly documented, and a proprietary, closed source compiler is used so that the software details are not open for inspection. This paper discusses the Epiphany-specific RPC implementation details in Sect. 3, the performance evaluation in Sect. 4, and a discussion of future work in Sect. 5.

3 Remote Procedure Call Technique

Although the RPC implementation described in this paper is designed for Epiphany, the techniques may be generally applicable to other PGAS architectures. The GNU Compiler Collection toolchain is used, without modification, to enable the RPC capability on Epiphany. In order to keep the interface as simple as possible, decisions were made to use a direct call method rather than passing function and arguments through a library call. This abstraction hides much of the complexity, but has some limitations at this time. The RPC dispatch routine (Algorithm 1) uses a global address pointer passed in the first function argument to select the remote core for execution. Up to four 32-bit function arguments may be used and are registers in the Epiphany application binary interface (ABI). In the case of 64-bit arguments, the ABI uses two consecutive registers. For the purposes of this work, any RPC prototype can work as long as the ABI does not exceed four 32-bit arguments and one 32-bit return value.

Algorithm 1. RPC dispatch routine

```
function RPC_DISPATCH($a1$,$a2$,$a3$,$a4$)
    rpc_call $\leftarrow$ $ip$
    $high\_addr \leftarrow mask(a1)$
    if $high\_addr = my\_addr$ then
        return RPC_CALL($a1$,$a2$,$a3$,$a4$)
    end if
    $remote\_lock \leftarrow high\_addr | lock\_addr$
    $remote\_queue \leftarrow high\_addr | queue$
    ACQUIRE_LOCK($remote\_lock$)
    if $remote\_queue$ is full then
        RELEASE_LOCK($remote\_lock$)
        return RPC_CALL($a1$,$a2$,$a3$,$a4$)
    end if
    $p\_event \leftarrow my\_addr | \&event$
    $remote\_queue$.push($\{a1, a2, a3, a4, rpc\_call, p\_event\}$)
    RELEASE_LOCK($remote\_lock$)
    if $remote\_queue$ initially empty then
        SIGNAL_REMOTE_INTERRUPT($high\_addr$)
    end if
    repeat
    until $event.status$ = rpc_complete
    return $event.val$
end function
```

An overview of the specialized RPC interrupt service request (ISR) appears in Algorithm 2. The user-defined ISR precludes other applications from using it, but it is sufficiently generalized and exposed so that applications can easily make use of it. The ISR operates at the lowest priority level after every other interrupt or exception. It may also be interrupted, but the RPC queue and the RPC dispatch method are designed so the ISR need only be signalled when the first work item is added to the queue.

Algorithm 2. RPC interrupt service request

```
function RPC_ISR
    ACQUIRE_LOCK($my\_lock$)
    while $queue$ is not empty do
        $work \leftarrow queue$.pop()
        RELEASE_LOCK($my\_lock$)
        $work.event.val \leftarrow work$.RPC_CALL($work.a1$,$work.a2$,$work.a3$,$work.a4$)
        $work.evevent.status$ = rpc_complete
        ACQUIRE_LOCK($my\_lock$)
    end while
    RELEASE_LOCK($my\_lock$)
end function
```

Setting up code to use the developed RPC methods is easy, but there are some restrictions. The subroutine should localize the global-local addresses to improve performance. A convenient inline subroutine has been made for this. If one of the addresses is non-local, it will not modify the pointer and default to RDMA. Listing 1 shows creating a dot product routine, a corresponding routine without a return value, and using the RPC macro to set up the RPC dispatch jump table.

```
float dotprod(float* a, float* b, int n)
{
        a = localize(a); // translate global addresses to local
        b = localize(b);
        float sum = 0.0f;
        for (int i = 0; i < n; i++) sum += a[i] * b[i];
        return sum;
}
RPC(dotprod) // dotprod_rpc symbol and jump table entry
```

List. 1. Example code for RPC function with symbol registration and jump table entry.

Using the RPC method is also very easy. The RPC calls made with the RPC macro have a `_rpc` suffix and are called like regular functions, but with the first pointer as a global address on a remote core. An example of application setup with mixed OpenSHMEM and RPC calls is presented in Listing 2. OpenSHMEM with symmetric allocation is not required, but it is convenient for demonstration.

```
float* A = shmem_malloc(n * sizeof(*A)); // symmetric allocation
float* B = shmem_malloc(n * sizeof(*B));
float* A1 = shmem_ptr(A, 1); // address of 'A' on PE #1
float* B1 = shmem_ptr(B, 1);
float res = dotprod_rpc(A1, B1, n); // RPC on PE #1
```

List. 2. Application code example for mixed usage of OpenSHMEM and RPC. The address translation for the B vector is necessary in case RDMA is required.

4 Results

The results presented include performance figures for an optimized single precision floating point RPC dot product operation using a stack-based RPC work queue. The dot product subroutine was chosen because it has a tunable work size parameter, n, relatively low arithmetic intensity at 0.25 FLOPS/byte—representing the challenging "data-intensive" operations—and reduces to a single 32-bit value result. Dot product performance for a single Epiphany core executing on local memory approaches the core peak performance of 1.2 GFLOPS and 4.8 GB/s. This corresponds to a dual-issue fused multiply-add and double-word (64-bit) load per clock cycle at 600 MHz. The small bump in the results around $n = 16$ on Figs. 1 and 2 is the result of the subroutine using a different code path for larger arrays. Figure 1 shows the total on-chip bandwidth performance for various execution configurations. The highest performance is achieved with a

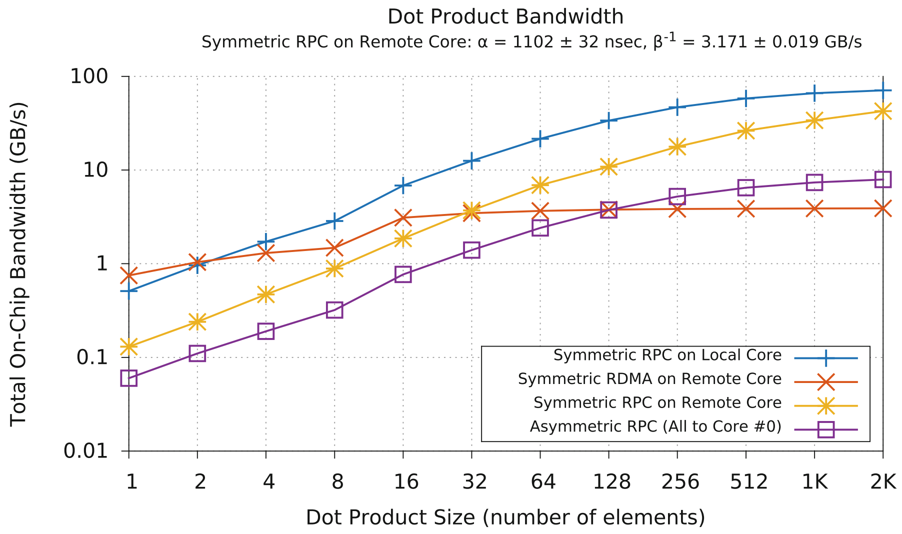

Fig. 1. Symmetric, or load-balanced, RPC with a stack-based queue can achieve over 60% of the bandwidth performance as if execution were accessing local scratchpad data (throughput up to 2.95 GB/s vs 4.8 GB/s per core). However, there must be sufficient work on the remote core to mitigate the overhead of the RPC dispatch versus RDMA.

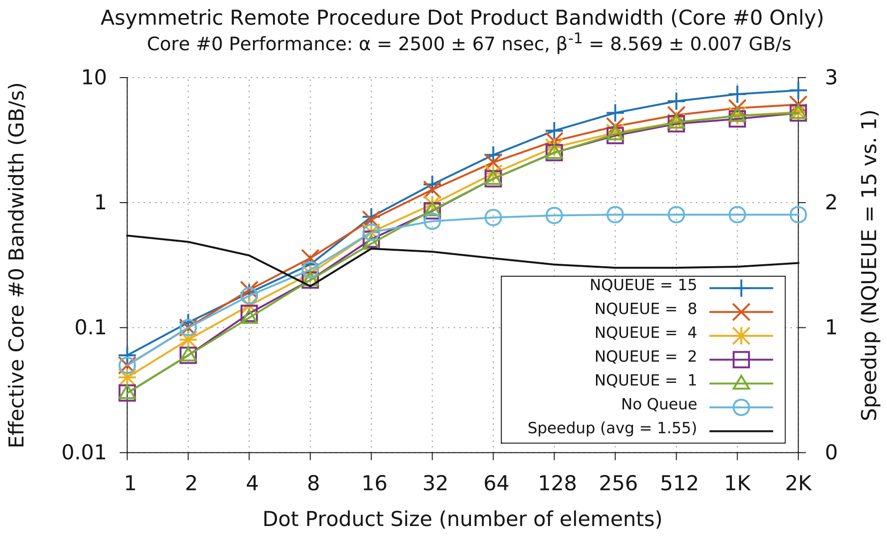

Fig. 2. Effect of RPC queue size for an asymmetric workload shows a speedup of about 1.5x from a queue size of one to 15. The greatest improvement comes from having a single extra workload enqueued (NQUEUE = 1) compared to no queue at all.

symmetric load executing a dot product on local data, without address translation (localization for array pointers *a* and *b* within the subroutine), which adds a small overhead.

A very positive and initially unexpected result occurred during the asymmetric loading for the RPC test where all 16 cores made requests to a single core (core #0). Peak performance of the operation was not expected to exceed 4.8 GB/s, but the timing indicated performance around 8 GB/s (Figs. 1 and 2). This is due to the local memory system supporting simultaneous instruction fetching, data fetching, and remote memory requests. If a remote work queue is filled with RPC requests, the calling core will perform RDMA execution as a fallback rather than waiting for the remote core resources to become available. This prevents deadlocking and improves overall throughput because no core is idle even if it is operating at lower performance. The result is that multiple cores may execute on data in different banks on a single core, effectively increasing bandwidth performance. Figure 2 shows the effect of increasing the RPC work queue size. There is no performance impact by increasing the queue size to more than the total number of cores on-chip minus one since each remote core will only add a single request to the queue then wait for completion.

5 Conclusion and Future Work

The combination of fast message passing with OpenSHMEM to handle symmetric application execution and the RPC techniques described here for handling asymmetric workloads remote procedure calls creates a very flexible and high-performance programming paradigm. This combination creates potential for good performance on diverse applications with both regular and irregular data layouts, memory access patterns, and program execution on the Epiphany architecture. We hope that developers on similar memory-mapped parallel architectures may use this paper as a guide for exploring the inter-processor RPC concept.

The developments in this paper will be built into the COPRTHR 2.0 SDK as low-level operating system services. It may also be used by some of the non-blocking subroutines in the *ARL OpenSHMEM for Epiphany* software stack for particular remote subroutines to enable higher performance. We will extend this work to support asynchronous RPC requests so programs do not block on remote operations. Additional reductions in instruction overhead may be found through low-level optimization of the RPC dispatch and software interrupt routines by transforming the high-level C code to optimized Epiphany assembly. The software interrupt appears to be overly conservative in saving the state and the dispatch method is overly conservative in assumptions of address locations and memory alignment, so these routines should be able to be substantially optimized. Since no considerations are made for load balancing and quality of service in this work, future development may allow for remote cores to defer servicing RPCs with tunable priority.

References

1. Olofsson, A., Nordström, T., Ul-Abdin, Z.: Kickstarting high-performance energy-efficient manycore architectures with Epiphany. In 2014 48th Asilomar Conference on Signals, Systems and Computers, pp. 1719–1726, November 2014
2. COPRTHR-2 Epiphany/Parallella Developer Resources. http://www.browndeertechnology.com/resources_epiphany_developer_coprthr2.htm. Accessed 01 July 2016
3. Ross, J., Richie, D.: An OpenSHMEM Implementation for the Adapteva Epiphany Coprocessor. In: Gorentla Venkata, M., Imam, N., Pophale, S., Mintz, T.M. (eds.) OpenSHMEM 2016. LNCS, vol. 10007, pp. 146–159. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50995-2_10
4. GitHub - US Army Research Lab/openshmem-epiphany - ARL OpenSHMEM for Epiphany. https://github.com/USArmyResearchLab/openshmem-epiphany/. Accessed 06 Feb 2018
5. Richie, D.A., Ross, J.A.: OpenCL + OpenSHMEM Hybrid Programming Model for the Adapteva Epiphany Architecture. In: Gorentla Venkata, M., Imam, N., Pophale, S., Mintz, T.M. (eds.) OpenSHMEM 2016. LNCS, vol. 10007, pp. 181–192. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50995-2_12
6. Richie, D., Ross, J., Infantolino, J.: A distributed shared memory model and C++ templated meta-programming interface for the epiphany RISC array processor. In: ICCS, 2017 (2017)
7. Chapman, B., Curtis, T., Pophale, S., Poole, S., Kuehn, J., Koelbel, C., Smith, L.: Introducing OpenSHMEM: SHMEM for the PGAS community. In: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, PGAS 2010, pp. 2:1–2:3. ACM, New York (2010)

8. Dysart, T., Kogge, P., Deneroff, M., Bovell, E., Briggs, P., Brockman, J., Jacobsen, K., Juan, Y., Kuntz, S., Lethin, R., McMahon, J., Pawar, C., Perrigo, M., Rucker, S., Ruttenberg, J., Ruttenberg, M., Stein, S.: Highly scalable near memory processing with migrating threads on the Emu System Architecture. In: Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms, IA$\hat{3}$ 2016, pp. 2–9. IEEE Press, Piscataway (2016)