Exploiting Communication Concurrency on High Performance Computing Systems

Nicholas Chaimov
University of Oregon
nchaimov@uoregon.edu

Khaled Z. Ibrahim, Samuel Williams, Costin Iancu
Lawrence Berkeley National Laboratory
{kzibrahim, swwilliams, cciancu}@lbl.gov

ABSTRACT

Although logically available, applications may not exploit enough instantaneous communication concurrency to maximize hardware utilization on HPC systems. This is exacerbated in hybrid programming models such as SPMD+OpenMP. We present the design of a "multi-threaded" runtime able to transparently increase the instantaneous network concurrency and to provide near saturation bandwidth, independent of the application configuration and dynamic behavior. The runtime forwards communication requests from application level tasks to multiple communication servers. Our techniques alleviate the need for spatial and temporal application level message concurrency optimizations. Experimental results show improved message throughput and bandwidth by as much as 150% for 4KB bytes messages on InfiniBand and by as much as 120% for 4KB byte messages on Cray Aries. For more complex operations such as all-to-all collectives, we observe as much as 30% speedup. This translates into 23% speedup on 12,288 cores for a NAS FT implemented using FFTW. We also observe as much as 76% speedup on 1,500 cores for an already optimized UPC+OpenMP geometric multigrid application using hybrid parallelism.

1. INTRODUCTION

Attaining good throughput on contemporary high performance networks requires maximizing message concurrency. As long as flat Single Program Multiple Data (SPMD) parallelism with one task per core has been dominant, this has not been a problem in application settings. Developers first employ non-blocking communication primitives and have multiple outstanding messages overlapped with other communication or computation inside *one* task. By using as many SPMD tasks as available cores, traffic is further parallelized over multiple injecting tasks within the node.

The advent of heterogeneous systems or wide homogeneous multicore nodes has introduced the additional challenge of tuning applications for intra-node concurrency, as well as communication concurrency. Manually tuning or transforming applications to provide the optimal message parallelism is difficult: 1) the right strategy is system dependent; 2) the right strategy is programming model dependent; and 3) parallelizing message streams may be complicated in large code bases. Furthermore, due to the implementation limitations described throughout this paper, to our knowledge optimizations to parallelize communication within a task have not been thoroughly explored.

We present the design of a runtime that is able to increase the instantaneous network concurrency and provide saturation independent of the application configuration and dynamic behavior. Our runtime alleviates the need for *spatial* and *temporal* application level message concurrency tuning. This is achieved by providing a "multi-threaded" runtime implementation, where dedicated communication server tasks are instantiated at program start-up along the application level tasks. We increase concurrency by forwarding communication requests from the application level to the multiple communication servers. This work makes the following contributions:

- We provide a detailed analysis of the optimization principles required for multi-threaded message injection. Our experiments on InfiniBand and Cray Aries networks indicate that saturation occurs differently based on the network type, the message distribution and the number of cores active simultaneously.

- We describe a runtime capable of maximizing communication concurrency *transparently*, without involving the application developer. The main insight is that after deciding the cores allowed to perform communication, one can attain saturation by solely using the message size to control the allocation of messages to cores.

- We quantify the performance benefits of parallelizing communication in hybrid codes and identify the shortcomings of existing runtime implementation practices.

Our implementation extends the Berkeley UPC [28, 5] runtime and therefore we demonstrate results for Partitioned Global Address Space (PGAS) applications and one-sided communication primitives. Experimental results indicate that our approach can improve message throughput and bandwidth by as much as 150% for 4KB messages on InfiniBand and by as much as 120% for 4KB messages on Cray Aries. Our runtime is able to transparently improve end-to-end performance for all-to-all collectives where we observe as much as 30% speedup. In application settings we observe

PMAM '15, February 7-8, 2015, San Francisco Bay Area, USA
Copyright 2015 ACM 978-1-4503-3404-4/15/02
http://dx.doi.org/10.1145/2712386.2712394 ...$15.00.

23% speedup on 12,288 cores for a NAS FT benchmark implemented in UPC+`pthreads` using FFTW [11]. We also observe as much as 76% speedup on 1,500 cores for an already heavily optimized UPC+OpenMP geometric multigrid [30] application using point-to-point communication.

We demonstrate performance benefits for hybrid programming using a PGAS programming language by exploiting shared memory within the node and one-sided communication. These characteristics are present in other models such as MPI 3 one-sided, as well as implementations of dynamic tasking languages such as X10 [7] or Habanero-C [13, 8]. Besides implicit parallelization, the principles presented apply to cases where communication is explicitly parallelized by the developer using OpenMP or other shared memory programming models.

The rest of this paper is structured as follows. In Sections 2 and 3 we discuss the design principles of multi-threaded message injection. In Section 4 we discuss network performance emphasizing the relationship between messge concurrency and bandwidth saturation. In Section 5 we discuss the integration of message parallelization into existing application settings. In particular we quantify the need for dynamic message parallelization and the impact of current core allocation mechanisms on performance. In Section 6 we summarize our results, while in Section 7 we present related work. We conclude the paper in Section 8.

2. COMMUNICATION AND CONCURRENCY

In order to actively manipulate message concurrency, program transformations must address both *spatial* and *temporal* aspects.

Spatial concurrency is controlled by choosing the number of active tasks (or cores) that perform communication operations, e.g. MPI ranks. By selecting a particular programming model, developers effectively choose the amount of spatial concurrency exploited within the application.

Temporal concurrency captures the insight that not all the tasks may want to communicate at the same time and the network may be perennialy under-utilized even when a large number of messages are logically available inside a task. Messages within a task are "serialized", even for non-blocking communication: 1) message injection is serialized inside the issuing task; and 2) parts of the message transmission may be serialized by the network hardware for any task. For load imbalanced or irregular applications, only few tasks may communicate at any given time and the message stream within any task could be further parallelized.

SPMD programs provide spatial concurrency by running one task per core. For well balanced applications, there usually exists communication concurrency, even enough to cause congestion [17]. In this case throttling the spatial concurrency of communication improves performance. To our knowldege temporal concerns have not been explored for load imbalanced SPMD codes.

Hybrid parallelism [6, 20] combines SPMD with another intra-node programming model such as OpenMP. Currently, communication is issued only from the SPMD regions of the code. When compared to pure SPMD, these new hybrid codes run with fewer "communication" tasks per node and consequently exhibit lower spatial concurrency. For example, hybrid MPI+CUDA codes [20, 19, 18] tend to use one MPI rank per GPU for programability and performance reasons. Hybrid MPI+OpenMP codes tend to use one MPI rank per NUMA domain for locality reasons. Previous work [31] showed that tuning the balance between the number of MPI ranks and OpenMP threads was essential in attaining best performance. Although that work suggested thread-heavy configurations were ideal for those machines (minimize data movement when a single thread can attain high MPI bandwidth), current machines (low MPI bandwidth per thread) can make a more nuanced trade between total inter-process data movement and total MPI bandwidth.

To our knowledge, techniques to further parallelize communication have not yet been shown beneficial in applications. As parallelizing the communication at the application level using OpenMP should be tractable, the main reason is the inability of current runtime implementations to provide good performance when mixing processes with `pthreads`. Communicating from OpenMP within one MPI rank requires running in `MPI_THREAD_MULTIPLE` mode, which has been reported [25] to negatively affect performance.

Applications written in programming models that support asynchronous task parallelism [8, 7] should offer the programmer high message concurrency, as every message can be performed inside an independent activity. However, this is mostly an illusion as communication is usually serialized inside the runtimes due to implementation constraints. For example, HCMPI [8] combines the Habanero-C [13] dynamic tasking parallel programming model with the widely used MPI message-passing interface. Inside the runtime there are computation and communication workers implemented as `pthreads`. To work around multi-threaded MPI's limitations, computation workers are associated with only one communication worker that uses MPI_THREAD_SINGLE. Thus, communication is de facto serialized inside a HCMPI program. X10 [7] implementations running PAMI on IBM BlueGene/Q can provide high message concurrency, but most likely serialize communication on non-IBM hardware.

In this work we argue for transparent parallelization of one-sided communication using a "multi-threaded" runtime implementation. We provide a decoupled parallel communication subsystem that handles message injection and scheduling on behalf of the application level "tasks". As this is designed to maximize network utilization, application level programmers need only to use non-blocking communication primitives without worrying about scheduling optimizations. While we show results for hybrid UPC+OpenMP and UPC+`pthreads` programming, these principles are applicable to other one-sided communication runtimes such as MPI-3 and map naturally into programming models using dynamic task parallelism such as Habanero-C. Accelerator based programming such as MPI+CUDA is another clear beneficiary of our approach.

From the above discussion, it is apparent that maximizing communication parallelism in programming models beyond SPMD faces several challenges. There is an engineering hurdle introduced by the requirement to mix processes with `pthreads` inside the runtime implementation. As *performance* is poor in most `pthreads` implementations[1], we explore a dual parallelization strategy using either processes or `pthreads` as communication servers. This approach is likely to be required for portability in the medium term future, as fixing `pthreads` on a per runtime basis is non-trivial.

[1]Exceptions are PAMI on IBM BG/Q and GASNet on Cray Aries.

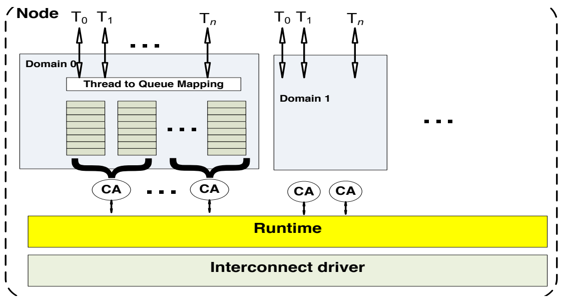

Figure 1: *Runtime architecture*

Transparent optimizations are good for programmer productivity, but one may argue that explicitly parallelizing communication using OpenMP is enough, were `pthreads` behaving well. Explicit manual communication parallelization faces *performance portability* challenges. First, performance is system dependent and it also depends on the instantaneous behavior of the application, i.e. how many tasks are actively communicating. Second, it is challenging to parallelize communication in an application already modified to overlap communication with other parallel computation.

3. RUNTIME DESIGN

Contemporary networks offer hardware support for one-sided Remote Direct Memory Access (RDMA) Put and Get primitives. Runtime implementations are heavily optimized to use RDMA and applications are optimized to overlap communication with other work by using non-blocking communication primitives of the form `{init_put(); ... sync();}`.

We target directy the UPC language [27], which provides a Partitioned Global Address Space abstraction for SPMD programming, where parts of the program heap are directly addressable using one-sided communication by any task. Our implementation is designed to improve performance of codes using the new UPC 1.3 non-blocking communication primitives, e.g. `upc_memput_nb(), upc_waitsync()`. We modify the Berkeley UPC implementation [5], which runs on top of GASNet [4]. GASNet provides a performance portable implementation of one-sided communication primitives.

Our idea is very simple: we achieve transparent network saturation by using a dedicated communication subsystem that spawns dedicated communication tasks, herein referred to as servers. Any communication operation within an application level task is forwarded to a server. Thus, we increase the parallelism of message injection by controlling the number of servers and we can control serialization deeper inside the network hardware by tuning the policy of message dispatch to servers.

The basic runtime abstraction is a communication domain. As shown in Figure 1, each communication domain has associated with it a number of clients (T_i) and a number of servers (CA). Any client can interact with any server within the same communication domain. In practice, communication domains are abstractions that can be instantiated to reflect the hardware hierarchy, such as NUMA domains or sockets. Clients are the application level tasks (threads in UPC parlance).

Servers are tasks that are spawned and initialized at program startup time. They provide message queues where clients can deposit communication requests. A communication request is a Put or Get operation and its arguments `(src, dest, size)`. While active, the server tasks scan the message queues, initiate and retire any requests encountered. In order to avoid network contention, servers can choose to initiate messages subject to flow control constraints, e.g. limit the number of messages in flight. To minimize interference with other tasks, servers are blocked on semaphores while message queues are empty.

To implement the client-server interaction we transparently redirect the UPC language-level communication APIs, *e.g.* `upc_memput_nb()` or `upc_waitsync()`, to our runtime and redefine the `upc_handle_t` datatype used for message completion checks. For any communication operation at the application level, our runtime chooses either to issue the message directly or to deposit a descriptor in one of the server queues. Both the order of choosing the next message queue and the number of messages deposited consecutively in the same queue are tunable parameters.

Any non-blocking communication operation returns a `handle` object, used later to check for completion. The client-server interaction occurs through messages queues, which are lock free data structures synchronized using atomic operations. In our implementation, application level communication calls return a value (`handle`) which represents an index into the message queues. Currently, we do not dynamically manage the message queue entries and clients have to explicitly check for message completion before an entry is reclaimed. This translates into a constraint at the application level that there is a static threshold for the number of calls made before having to check for message completion.

The UPC language allows for relaxed memory consistency and full reordering of communication operations. This is the mode used in practice by applications and our servers do not yet attempt to maintain message ordering. Strict memory consistency imposes order on the messages issues within a UPC thread. In our implementation this is achieved by having the initiator task perform the strict operations directly.

3.1 Implementation Details

Achieving performance requires avoiding memory copies and maximizing the use of RDMA transfers, which at the implementation level translates into: 1) having shared memory between tasks; 2) having memory registered and pinned in all tasks; and 3) having tasks able to initiate communication on behalf of other tasks. We provide a dual implementation where servers are instantiated as either processes or `pthreads`. The underlying UPC runtime implementation provides shared memory between tasks in either instantiation.

Previous work [3] indicates that best RDMA communication performance in UPC is attained by process-based runtime implementations, i.e. the applications run with one process per core. As this is still valid[2] for most other runtimes on most existing hardware, our first prototype spawned servers as standalone processes inside the runtime. This required non-trivial changes to the BUPC runtime. However, as discussed later, idiosyncrasies of existing system software determined us to provide a `pthreads`-based implementation for scalability reasons. The use of shared memory within multicore node, for instance using OpenMP, allows less replicated state [2] and reduces the memory usage of runtime, which is critical at scale. While our process-based implemen-

[2]Except PAMI on IBM BG/Q, GASNet on Aries.

tation requires modified UPC runtime, the `pthreads` is written using unmodified UPC runtime and can be distributed as a stand-alone portable library. Furthermore, the latter implementation can take advantage off the good `pthreads` performance of GASNet [14] on Cray GNI messaging library (supported on Gemini and Aries interconnects).

Startup: RDMA requires memory to be pinned and registered with the network by any task involved in the operation. `pthreads` inherit registration information from their parent processes, thus servers as `pthreads` can be spawned at any time during execution, including user level libraires.

Getting both shared memory and registration working together with servers as processes required complex modifications to the Berkeley UPC runtime code. The BUPC startup code initializes first the GASNet communication layer and then proceeds to initialize the shared heap and the UPC language specific data structures. As RDMA requires memory registration with the NIC, having the communication servers as full fledged processes requires them to be spawned at job startup in order to participate in any registration sequence.

Tasks spawned by the job spawner are captured directly by GASNet. Inside the UPC runtime there exists an implicit assumption that any task managed by GASNet will become a full-fledged UPC language thread. Furthermore, there is little control over task placement and naming as enforced by the system job spawner. We had to modify the UPC startup sequence to intercept and rename all server tasks before the UPC specific initialization begins. This cascaded into many other unexpected changes imposed by the BUPC software architecture. Internally, we split the UPC and server tasks into separate GASNet teams (aka MPI communicators) and reimplement most of the UPC runtime APIs to operate using the new task naming schema. In particular, all UPC pointer arithmetic operations, communication primitives, collective operations and memory allocation required modifications.

RDMA and Memory: The new UPC 1.3 language specification provides the `upc_castable` primitive to allow passing of addresses between tasks within a node. `pthreads`-based implementation can perform RDMA on these addresses, albeit with performance loss. Shared addresses are not guaranteed to be legal RDMA targets when passed between processes. GASNet registers at startup memory segments for each known process. Only the process that has explicitly registered the segment can use RDMA on that region. Thus, one solution is to use statically duplicate registration of all application memory segments inside all servers. Another solution is to use dynamic registration inside servers. For unregistered addresses, GASNet uses internally an algorithm that selects between memory copies into bounce buffers for small messages or dynamic registration for large messages. A similar approach [24] is used internally inside MPI implementations.

Duplicate registration required breaking software encapsulation and extending the modifications from the UPC language runtime all the way to GASNet, which aims to be a language independent communication library. Instead, we chose to exploit the dynamic registration mechanism in GASNet. This turned out to be a fortuitous design decision as some underlying communication libraries (Cray uGNI) did not allow unconstrained registration of the same memory region in multiple processes.

Synchronization: The base GASNet implementation requires that communication operations are completed by the same task that has initiated them with the network. This constraint necessitates special handling of non-blocking communication primitives in our runtime. We introduced an extra synchronization step for message completion between clients and servers. Removing this constraint will require a significant redesign of GASNet communication infrastructure, which is not warranted by the observed performance. Furthermore, note that achieving good performance in practice required a careful tuning of atomic operations usage and runtime data structures padding to avoid false sharing.

Although it appears that these restrictions and design decisions are particular to the Berkeley UPC and GASNet implementations, most existing runtimes use similar software engineering techniques. As recently shown [14], combining MPI with `pthreads` still leads to performance degradation. We expect that trying to parallelize communication over processes while preserving shared memory similar in a different code base will encounter the same magnitude problems. Retrofitting spawning separate processes to act as communication servers into an existing runtime is likely to require coordinated changes across all abstraction layers.

4. NETWORK PERFORMANCE AND SATURATION

Performance when using non-blocking communication is determined by the number of cores active within the node, as well as the number of outstanding messages per core. Our microbenchmark takes measurements for different numbers of cores active and reports the percentage of the peak *bi-directional* bandwidth attained at a particular message size and messages per core. The peak attainable bandwidth for a message size is determined as the maximum bandwidth observed across all possible combinations (cores, messages per core) at that size.

In Figures 2 and 3 (top) we present the performance of the Berkeley UPC [5] compiler running on the GASNet [4] communication layer. We report results for InfiniBand and the Cray Aries networks when each task is instantiated as a OS level process. `pthreads` are omitted for brevity, they match [14] process performance on Aries and are significantly slower in InfiniBand.

Edison: is a Cray XC30 MPP installed at NERSC[3]. Each of its 5200 nodes contains two 12-core Ivy Bridge processors running at 2.4 GHz. Each processor includes a 20 MB L3 cache and four DDR3-1866 memory controllers which can sustain a stream bandwidth in excess of 50 GB/s. Every four nodes are connected to one Aries network interface chip. The Aries chips form a 3-rank dragonfly network. Note that depending on the placement within the system, traffic can traverse either electrical or optical links. While the attainable bandwidth is different, all other performance trends of interest to this study are similar for both link types.

Figure 2 (top) presents the results on Edison for a four node experiment (two NICs). Put operations are usually faster than Get operations, by as much as 25% for medium to large messages. For small to medium messages, Put op-

[3]National Energy Research Scientific Computing Center.

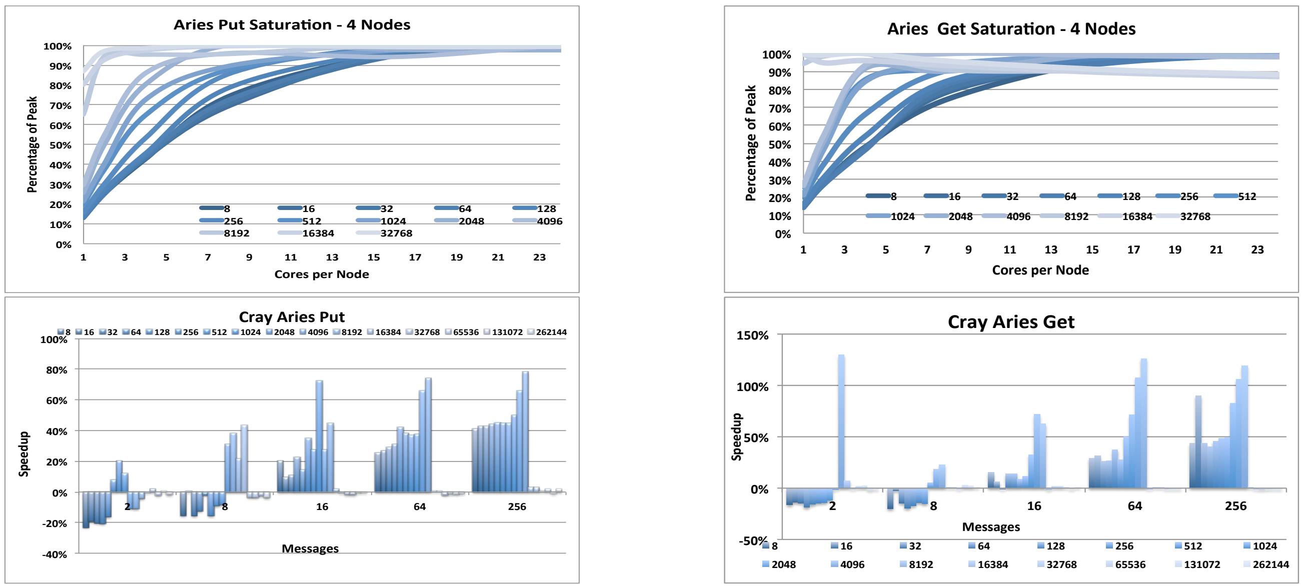

Figure 2: *Top: Cray Aries saturation, four nodes, 24 cores per node. Bottom: Performance improvements on Cray Aries with message size, number of messages. Experiment uses all sockets within node, one rank per socket, two servers per socket. Policy is round-robin of four messages to a server. Only small to medium messages benefit from parallelization, as indicated by the saturation graph.*

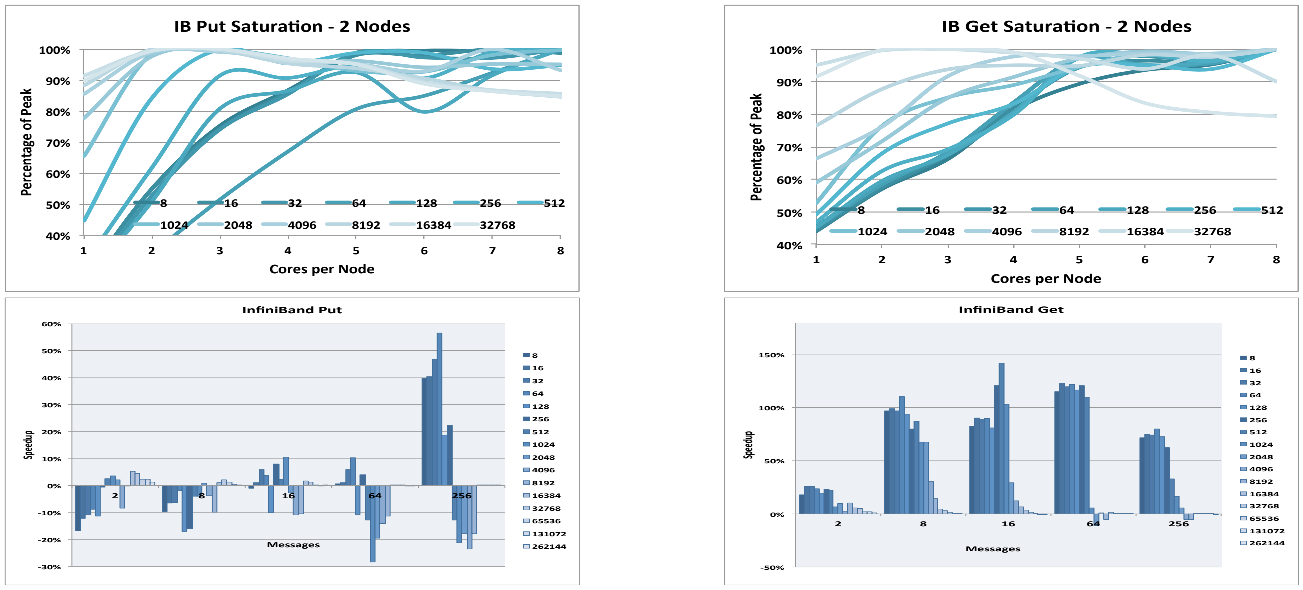

Figure 3: *Top: InfiniBand saturation, two nodes, eight cores per node, two sockets. Bottom: Performance improvements of InfiniBand with message size, number of messages. All sockets active, two servers per socket. Only small to medium messages benefit from parallelization, as indicated by the saturation graph.*

erations need more cores than Get operations to reach saturation. For example, for 1024 byte messages, Puts require more than eight cores, while Gets require only four cores. For large messages, saturation is reached with only one core active. Increasing the number of active cores determines a bandwidth decrease for large messages.

Carver: is an IBM Infiniband cluster installed at NERSC. Each of its nodes contains two 4-core Xeon X5550 (Nehalem) processors running at 2.67 GHz. Each processor includes a 8 MB L3 cache and three DDR3-1333 memory controllers which can sustain a stream bandwidth of up to 17.6 GB/s. Nodes are connected via QDR InfiniBand using a hybrid (local fat-tree/global 2D mesh) topology.

Figure 3 (top) presents the experimental results for two nodes. For small to medium messages, Put operations are up to 8X faster than Get operations, For "medium" messages we observe a 3X bandwidth difference for 512 byte messages. For Put operations, it takes four or more cores to saturate the bandwidth for messages shorter than 512 bytes. For larger messages, one or two cores can saturate the network, as illustrated for 32KB messages. Get operations saturate the network slower than Put operations, and it takes four or more cores to saturate for messages smaller that 8KB. For both operations, increasing the number of active cores for large messages decreasees performance by up to 20% for 32KB messages.

Both networks exhibit common trends that illustrate the challenges of tuning message concurrency:

- Put and Get operations exhibit different behavior on the same system and across systems. Optimizations need to be specialized per operation, per system.
- For small to medium messages, bandwith saturation occurs only when multiple cores are active with multiple outstanding messages. Parallelization is likely to improve performance in this case.
- For medium to large messages, bandwidth saturation occurs with few cores per node, may degrade when increasing the number of cores per node. Parallelization may degrade performance in this case.

4.1 Evaluation of Parallelization

We evaluate the performance of our approach on the same microbenchmark in settings with one UPC thread per node and with one UPC thread per NUMA domain. The former is the typical setup in distributed applications that use GPUs. The latter is the setup used in manycore systems when mixing distributed and shared memory programming models.

We vary the number of server tasks from one to the number of cores available in the NUMA domain. We consider two strategies for message forwarding. In the first approach, clients forward communication in a round-robin manner to servers and also actively initiate some of their communication operations, similar to a hybrid SPMD+X configuration. In the second approach clients are inactive and forward all operations to servers in a round robin manner, similar to a dynamic tasking configuration such as HCMPI. Another tuning parameter is the number of operations consecutively forwarded to one server, which we vary from one to ten.

In our experiments the communication domains are confined within the same NUMA domain as their clients. We are interested in determing the optimal software configuration for our runtime which includes: 1) the number of servers per communication domain and NUMA domain; 2) order of choosing a server; 3) the message mix assigned to a server at any given time.

4.1.1 Cray Aries

The Cray Aries network provides two mechanisms for RDMA: Fast Memory Access (FMA) and Block Transfer Engine (BTE). FMA is used for small to medium transfers and works by having the processors writing directly into a FMA window within the NIC. The granularity of the hardware request is 64 bytes. BTE is employed for large messages. The processor writes a transfer descriptor to a hardware queue and the Aries NIC performs the transfer asynchronously. BTE supports up to four transfers. Communication APIs written on top of the Cray GNI or DMAPP system APIs switch between FMA and BTE for transfers in the few KB range. For GASNet the protocol switch occurs at 4KB.

GASNet [14] has been thoroughly re-engineered recently to provide good performance with `pthreads` on Cray systems. Figure 2 (bottom) shows the performance improvements for this instantiation of our server code. Most of the improvements of parallelization are directly correlated with the saturation graph in the same Figure 2. We observe similar behavior when one or both sockets within the Cray nodes are active.

Parallelization does not seem to help much when there are fewer than four or eight messages available at the application level. For longer message trains parallelization does help and we observe speedups as high as 130%.

Medium size messages benefit most at a low degree of parallelization, smaller messages require more servers. This is correlated with the saturation slope in Figure 2. For example parallelizing with two servers 64 Gets each of size 4096 bytes yields a 120% speedup, while parallelizing 64 eight byte operations yields only a 30% speedup. Paralellization does not yield great benefits for transfers larger than 4KB. This indicates that for this traffic pattern BTE transfers do not benefit from it.

We omit detailed results for the process based implementation. Transfers smaller than 8KB can be parallelized, while in our implementation larger messages are issued directly by the clients. When pointers to messages larger than 8KB are passed to a server process, GASNet switches to dynamic registration and the Cray uGNI library disallows registration of the same GASNet memory region into multiple processes. We have also experimented with using bounce buffers inside servers for large transfers without any worthwhile performance improvements.

Overall, `pthreads` based parallelization works very well on Cray Aries, while process based parallelization does not.

4.1.2 InfiniBand

On InfiniBand, parallelization using `pthreads` severely degrades performance, while parallelization over processes improves it.

Figure 3 (bottom) shows performance results on the InfiniBand system when using processes for parallelization. Overall, best performance results are obained for small to medium messages, up to 4KB, which require multiple cores to saturate the network. Larger messages saturate with only a few cores and should not benefit from parallelization. Furthermore, when passing an address between processes, the underlying GASNet implementation chooses between RDMA using bounce buffers for messages smaller than a page and in-place RDMA with dynamic memory registration for larger transfers. Dynamic memory registration requires system calls which serialize the large transfers. Note that this combination of bounce buffers and dynamic registration also reduces the performance benefits of parallelization.

Parallelization provides best results for Get operations which saturate the network slower than Puts. In the best configuration we observe as much as 150% speedup from parallelization for 4KB messages. The technique is effective for Gets even when very few operations (as low as two) are available. For Gets, increasing the degree of parallelization improves performance and best performance is obtained when using most cores within the NUMA domain.

In the case of Puts the best speedup observed is around 80% for 128 bytes messages and the technique requires at least 32 messages per thread before showing performance improvements when using one socket. For Puts, increasing the degree of parallelization does not improve performance, as illustrated in Figure 3 bottom left.

Again, understanding the saturation behavior is a good indicator for the benefits of parallelization of communication.

5. PARALLELIZING INJECTION IN APPLICATIONS

Although the perfomance improvements are certainly encouraging for regular communication behavior, applications may exhibit instantaneous behavior which is adversary to our approach.

In some settings the message mix may be unpredictable and there may exist resource contention between servers and computation tasks. To handle message mixes we have implemented a dynamic parallelization of injection using a performance model that takes into account the expected number of messages, message size and type. To handle core contention we experiment with both "cooperative" scheduling and resource partitioning. All these mechanisms are exposed at the application level through a control API.

We experiment with a UPC+OpenMP multigrid benchmark we developed specifically for this study, as well as a 3D fast Fourier Transformation using the multithreaded FFTW [11] library and all-to-all collective communication.

Selective Parallelization: In order to provide optimal performance we need to know the number of messages, their size and type (put or get). We can then decide if parallelization improves performance and if so, we need to decide the optimal number of servers and message injection policy.

Based on the microbenchmark results, for any message size we determine a threshold on the number of messages to enable parallelization. For example, on Aries parallelization should be enabled any time there are more than four Get messages of size smaller than 8KB. For large messages we provide a direct injection policy by client tasks, bypassing the servers entirely. Any message larger than 8KB is directly injected by the client in our Aries implementation. As the actual number of messages does not matter, we provide application level APIs to simply enable and disable parallelization.

Once parallelization is enabled we need to choose the number of servers. Based on the experimental data, the optimal point is different for small and medium messages: small messages require more parallelism, medium messages less. On the other hand, for a fixed number of servers, the instantaneous message concurrency is actually determined by the injection policy. By simply varying the number of consecutive messages assigned to a server, we can directly control their concurrency: the larger this number, the lower the concurrency.

In our implementation, we allow developers to specify a static concurrency for the communication subsystem, based on core availability or application knowledge. For a specific concurrency, we build a control model that decides how many consecutive messages of a certain size are assigned to a server queue, e.g. we assign every other small message to a new server and increase this threshold with message size.

Note that the required server concurrency depends whether the clients can be active or need to be inactive, as determined by the programming model. Same heuristics apply in both cases.

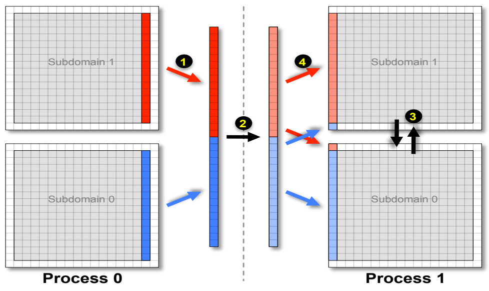

Figure 4: *A 2D visualization of the exchange boundary communication phase among two neighboring processes each with two subdomains. Note, only one direction (of 6) is shown. Only sends from process 0 are shown.*

Core Management: The dedicated communication subsystem may run concurrently with computation tasks. In this case the cores may be oversubscribed with computation and communication tasks and performance is also determined by the core allocation. We explore both cooperative scheduling approaches as well as partitioning approaches. For cooperative scheduling, communication tasks in idle states are sleeping and we provide interfaces to explicitly `wind-up` and `wind-down` these tasks. We experiment with different strategies: 1) best-effort, no task pinning; 2) pinning the communication tasks to core domains; 3) partitioning cores between communication and computation tasks and parallelizing all transfers; and 4) partitioning cores between computation and communication tasks and doing selective parallelization.

5.1 miniGMG

Multigrid is a linear-time approach for solving elliptic PDEs expressed as a system of linear equations ($Lu^h = f^h$). That is, MG requires O(N) operations to solve N equations with N unknowns. Nominally, MG proceeds by iterating on V-Cycles until some convergence criterion is reached. Within each V-Cycle, the solution is recursively expressed as a correction arising from the solution to a smaller (simpler) problem. This recursion proceeds until one reaches a base case (coarse grid) at which point, one uses a conventional iterative or direct solver. Multigrid's recursive nature states that at each successively coarser level, the computational requirements drop by factors of 8×, but the communication volume falls only by factors of 4×. As a result, multigrid will see a wide range of message sizes whose performance is critical to guaranteeing multigrid's O(N) computational complexity translates into an O(N) time to solution.

miniGMG is a small (3 thousand lines of C), publicly-available benchmark developed to proxy the geometric multigrid solves within the AMR MG applications [21, 30]. Geometric multigrid (GMG) is a specialization of multigrid in which the PDE is discretized on a structured grid. When coupled with a rectahedral decomposition into subdomains (boxes), communication becomes simple ghost zone (halo) exchanges with a fixed number of neighbors. In miniGMG, communication is performed by the MPI ranks, while all computation is aggressively threaded using OpenMP.

For this paper, using the publicly-available MPI+OpenMP implementation, we developed several UPC+OpenMP variants using either *Put* or *Get* communication paradigms with either barrier or point-to-point synchronization strategies. We only report results using the *Get* based implementation with point-to-point synchronization as it provides the best performance in practice. When compared to the original MPI+OpenMP version, our variant always provides matching or better performance.

In order to minimize the number of messages sent between any two processes, miniGMG's ghost zone exchange was optimized to aggregate the ghost zones exchanges of adjacent subdomains into a single message. Thus, as shown in Figure 4, two subdomains collocated on the same node will: 1) pack their data into an MPI send buffer; 2) initiate an MPI send/recv combination; 3) attempt to perform a local exchange while waiting for MPI; and 4) extract data from the MPI receive buffer into each subdomains private ghost zone. In each communication round a MPI rank exchanges only six messages with its neighbors. While this approach to communication is common place as it amortizes any communication overheads, it runs contrary to the need for parallelism. The UPC+OpenMP implementation we report uses the same algorithm.

As real applications use a variety of box sizes to balance AMR and computational efficiency with finite memory capacity and the desire to run physically realistic simulations, we evaluate performance using box sizes of 32^3, 64^3 and 128^3 distributed as one, eight or 64 boxes per UPC thread for both communication strategies. Some of the larger configurations will be limited by on-node computation, while smaller problems will be heavily communication-limited. Overall, due to varying degrees of required parallelism, aggressive message aggregation optimizations and different message sizes miniGMG provides a realistic and challenging benchmark to message parallelization.

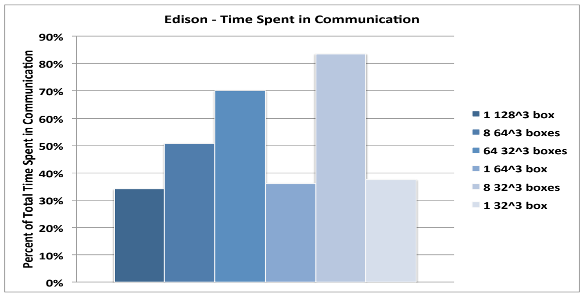

Figure 5: *Fraction of time spent doing communication in miniGMG on Edison is heavily dependent on the number of boxes per process (independent messages) rather than total data volume.*

miniGMG Performance: Figure 5 presents the fraction of our UPC miniGMG solve time spent in communication for a variety of problem and box sizes. As illustrated, the code can transition from computation-dominated to communication dominated with a sufficient number of boxes per process.

Both OpenMP threads and our communication server tasks require cores to run on. Figure 6 (left) presents the impact of using three communication threads compared to the baseline UPC+OpenMP implementation. In both cases, there are 12 OpenMP threads, but in the latter, the operating system must schedule the resultant 15 threads on 12 cores.

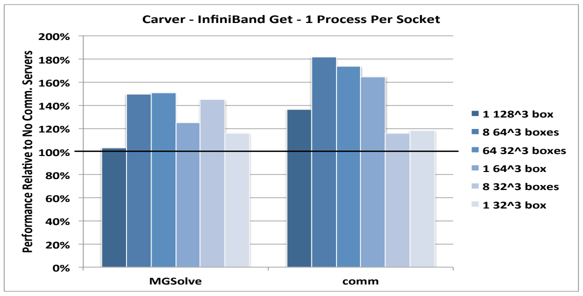

Figure 8: *Performance of UPC miniGMG with communication servers relative to UPC miniGMG without, on InfiniBand, with 2 client processes per node (1 per socket).*

MGSolve records the overall speedup on the multigrid solver while *comm* records the speedup in the communication operations. Inside the benchmark we explicitly use cooperative scheduling for the communication subsystem, i.e. communication tasks are sleeping when not needed. No thread is explicitly pinned and we have experimented with different OpenMP static and dynamic schedules. As illustrated, for all problem settings we observe performance degradation up to 25%.

Rather than oversubscribing the hardware and giving the scheduler full control to destroy any cache locality or to delay message injection, we experimented with eliminating oversubscription and pinning just the communication tasks. In this case we use 8 OpenMP threads and 3 pinned communication tasks. Although superior to oversubscription, performance is still less than the baseline. Detailed results are omitted.

Figure 6 (right) presents the speedup when hardware resources are partitioned among 8 OpenMP threads and 3 communication threads. Both OpenMP and communication threads are explicitly pinned to distinct cores and all communication is parallelized. Some problems observe substantial speedups (by as much as 70%), while some slow down by as much as 47%. On average we observe 2% slowdown and any performance degradation is explained by slowdown in communication.

Figure 7 (left) presents the best performance attained using selective parallelization at its optimal setting in a partitioned node. We now observe performance improvements for all problem settings, with a maximum of 76% and an average improvement of 40%. Figure 7 (center) shows results for selective parallelization using the adaptive strategy with two servers. Figure 7 (right) shows results of the adaptive strategy with three servers, giving a maximum improvement of 64% and an average improvement of 36%. As illustrated, allocating more cores to the communication subsystem improves performance and the adaptive strategy provides most of the possible performance gains.

For brevity we did not present detailed results on InfiniBand, they are similar to the results presented on the Cray system. Figure 8 shows an experiment with partitioned resources and parallelization enabled over three servers. Again we observe application speedup up to 80%.

These results indicate that under the current OS scheduling techniques, parallelizing communication successfuly requires partitioning and pinning.

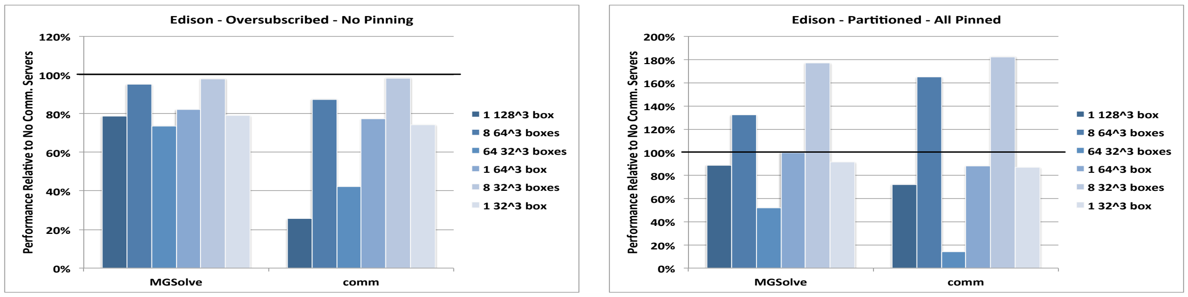

Figure 6: *Performance of UPC miniGMG with parallelization relative to UPC miniGMG without on Cray Aries. Left: Oversubscribed, best effort 12 OpenMP tasks, 3 servers on 12 cores. Right: paritioned, pinned, 8 OpenMP tasks, 3 servers on 12 cores. Best performance requires partitioning and explicit pinning of all tasks. Parallelization results in performance improvements for some problem sizes.*

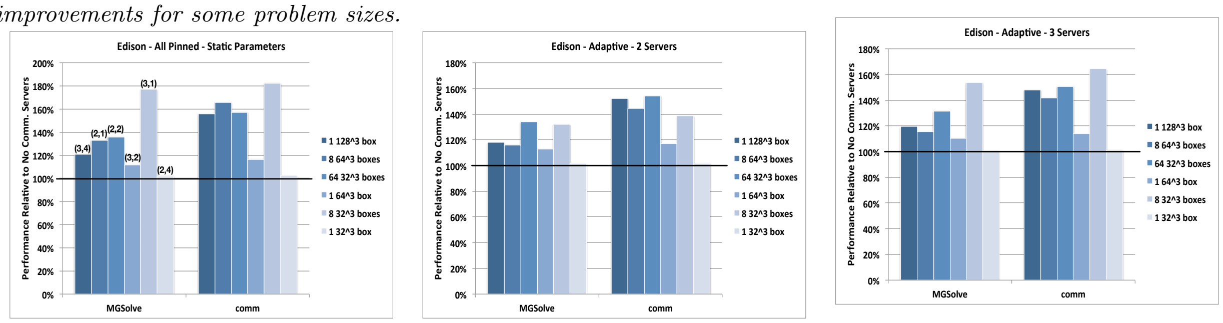

Figure 7: *Performance of UPC miniGMG with* **selective parallelization** *relative to UPC miniGMG without on Cray Aries. Left: optimal settings (server, batch size) are annotated on the figure. Center and right: adaptive parallelism with two and three servers. Allocating more cores to communication improves performance.*

5.2 Collective Operations

Optimizing the performance of collective operations [26, 15, 33] has seen its fair share of attention and implementations are well tuned by system vendors. Due to their semantics, collectives are an obvious beneficiary of our techniques in application settings as they mostly require tasks to contribute equal amount of data to a communication pattern with a large fan-out.

In Figure 9 we show the aggregate bandwidth of an `all-to-all` operation implemented using UPC one-sided *Get* operations[4] with and without parallel injection on 1,024 nodes of Edison, accounting for 12,288 total cores in a hybrid setting. Our implementation initiates non-blocking communication in a loop and throttles the number of outstanding messages to 128 for scalability with nodes. Parallelizing injection improves performance up to 30% over the baseline UPC case for messages smaller than 4KB.

For reference we include the performance of the Cray tuned `MPI_alltoall`. This implementation selects different algorithms for small (Bruck's algorithm) and large (pairwise exchange) messages, while our microbenchmark uses a single algorithm for all message sizes. Parallel injection allows our implementation to provide greater bandwidth in the region of messages sizes where MPI and our implementation use similar algorithms. Performance is better than the MPI version for messages between 8B and 2KB. On a smaller (64-node) run, performance was better than MPI for messages between 16B and 32KB.

We observe similar performance improvements up to 30% on InfiniBand, detailed results omitted for brevity. We expect to see similar trends while parallelizing other operations such as reductions and broadcasts.

[4]Note that this implementation provides better performance than a *Put* based implentation.

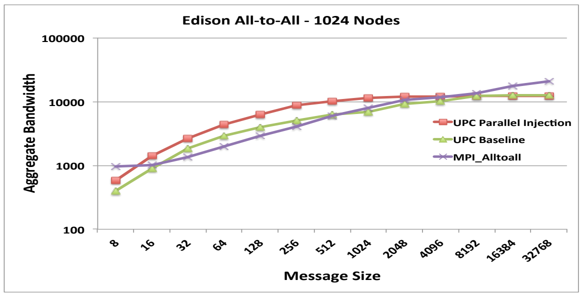

Figure 9: *Aggregate bandwidth achieved with one-sided put and get operations using UPC without parallel injection, UPC with parallel injection, and MPI.*

5.3 NPB UPC-FT

This benchmark implements the NAS Parallel Benchmarks [1] discrete 3D Fast Fourier Transform, using UPC for inter-node transpose communication and multi-threaded FFTW [11] for intra-node parallelization [29]. UPC-FT goes through two rounds of communication. For a problem of size $N_X \times N_Y \times N_Z$ run on a $P_X \times P_Z$ process grid, messages are $16 \cdot {}^{N_X}/_{P_X} \cdot {}^{N_Y}/_{P_X}$ bytes in the first round and $16 \cdot {}^{N_Y}/_{P_Z} \cdot {}^{N_X}/_{P_X}$ bytes in the second round.

Figure 10 shows the relative performance of UPC-FT on a class A size ($256 \times 256 \times 128$) problem on 1,536 cores of Edison, with the partitioning of the problem across nodes varied to produce first-round message sizes from 256B to 256KB while holding second-round message size constant at 8KB. FFTW is build with threading support using OpenMP

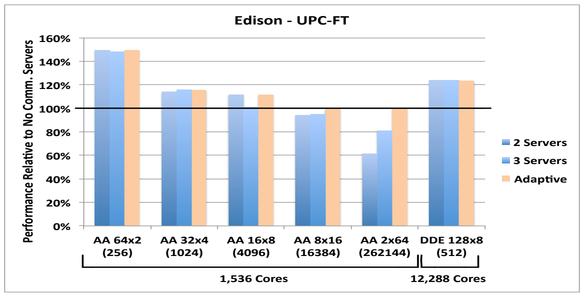

Figure 10: *Performance of the UPC-FT benchmark with Class A problem sizes on 1,536 cores of Edison for different first-round message sizes with two or three communication servers, relative to performance without communication servers, and for a class D-$^1/_8$ problem size on 12,288 cores of Edison.*

and configured to use 8 threads per process. OpenMP and communication servers threads are pinned to cores. The "2 Servers" and "3 Servers" columns show the performance effect of using that number of communication servers and parallelizing everything, while "Adaptive" shows the performance with selective parallelization.

Speedups of up to 49% are seen for the smallest messages, they decrease with increasing message sizes, with speedups of 11% for 4KB messages. Incidentally, the best original performance is obtained for the AA 32 × 4 setting. The rightmost section of the figure shows results on a class D-$^1/_8$ size (1024 × 512 × 512) problem distributed across a 128 × 8 process grid on 12,288 cores of Edison, with first-round message sizes of 512B. A speedup of 23% is achieved on this problem used by NERSC for system procurements.

6. DISCUSSION

Figure 11 summarizes the overall performance trends uncovered by this work. On the left hand side we compare the performance of a setting with one task per NUMA domain (hybrid parallelism in application) with our parallel injection. Parallelization occurs over two servers and for reference we include the peak bandwidth attainable on the system in any combination.

For both systems parallelization is effective for small to medium messages, up to 8KB on Aries and 32KB on InfiniBand. Parallelization does not improve the performance for large messages. For any message size, there is a gap between the parallelized injection and the *peak* attainable bandwidth. Most of this gap is accounted by injection concurrency and not by our implementation overhead.

For small to medium messages increasing the number of servers in 'PAR' closes the gap between attained and peak performance. For large messages, parallelization does not improve performance when compared to the original setting, yet there is a noticeable difference from peak bandwidth. In this case orthogonal concurrency throttling techniques as described by Luo [17] are required. Note that due to decoupling the communication into a standalone subsystem, these techniques are easy to implement in our architecture.

The right hand side graph in Figure 11 illustrates an intriguing opportunity. Medium messages at high concurrency achieve similar bandwidth to the best bandwidth achieved by large messages at any concurrency. This means that concurrency throttling or flow control techniques for large messages may be replaceable by message decomposition and parallel injection. We are currently investigating this tradeoff in our current infrastructure.

Overall our work makes the case for decoupling communication management from the application itself and transparently applying injection parallelization in conjunction with throttling in order to maximize throughput. Having a separate communication subsystem enables dynamic management on a node wide basis. This architecture fits naturally in both SPMD and dynamic tasking runtimes such as Habanero-C or HCMPI.

In our experiments, dedicating cores to communication affected only marginally, if at all, the end-to-end benchmark performance. Furthermore, for any problem where communication was present, its parallelization provided by far the best performance. We believe that dedicating a small number of cores to communication is feasible for many applications on existing systems. Of course, there may be computationally intensive applications that perform very little communication or synchronization.

Hardware evolutionary trends are also favorable to a decoupled parallel communication subsystem in application settings. There is likely to be enough core concurrency that a runtime system can instantiate a partition dedicated to communication management. This avoids scheduling problems when cores are oversubscribed. There also exists an expectation that in future systems the memory per core will decrease while the number of nodes will significantly increase. This implies that hybrid parallelism algorithms will have to use a small number of "traditional" communication tasks per node due to memory scalability problems inside runtimes (connection information), as well as the application levels (boundary conditions buffer space).

For hybrid programming such as UPC+OpenMP, it may seem that one can just fix `pthreads` and retrofit the principles we describe inside the applications themselves. The caveat is that the requirement to have both process and `pthreads`-based implementations for portability is unlikely to disappear in the foreseeable future. The first hybrid MPI+OpenMP studies [6] were published circa 2000. Fixing `pthreads` is not easy as illustrated by the performance in 2014. Furthermore, the low-level networking APIs and system software make this distinction necessary for performance portability, and unlikely to change.

7. OTHER RELATED WORK

As already explained in Section 2, explicit communication parallelization for hybrid SPMD+{OpenMP,CUDA} codes has not been thoroughly explored due to implementation constraints. Rabenseifner et al [23] discuss its potential and implications on algorithm design, without detailed performance results. Similarly, communication parallelization has not been yet explored in dynamic tasking runtimes. There has been work inside the MPI implementation [10, 12] to improve performance for MPI_THREAD_MULTIPLE. These studies demonstrate improved performance only for microbenchmarks, mostly on IBM BG/P hardware. Recent work by Luo et al [16] describes an MPI implementation able to provide improved performance for hybrid MPI+OpenMP parallelism on InfiniBand networks. They use multiple endpoints for parallelism and show results for microbenchmarks and

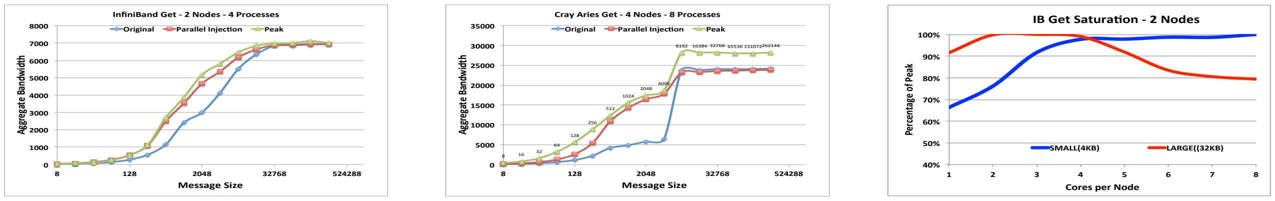

Figure 11: *Overall trends comparing hybrid setup with 1 task per NUMA domain('Original'), with parallelization ('PAR') and peak attainable bandwidth on the system. 'PAR' uses only 2 servers. Right - InfiniBand saturation with active cores for small and large messages.*

all-to-all operations. Dinan et al [9] discuss extensions to improve MPI interoperability with other programming models and `pthreads`. No performance results are presented in the study. Until performance portability is provided, developers are unlikely to adopt explicit parallelization in their codes.

Multi-threading the runtime implementation has been explored for both one-sided and two-sided communication paradigms. Recent efforts by Si et al [24] examine multi-threading the MPI runtime implementation. This implementation uses OpenMP multi-threading to accelerate internal runtime routines such as buffer copying and derived datatypes, while maintaining the conventional serialized message injection. They report a tight integration of MPICH with the Intel OpenMP runtime and demonstrate results only for shared memory programming on a single Intel Xeon Phi. ARMCI [22] implements a portable one-sided communication layer that runs on most existing HPC platforms and uses `pthreads` for network attentiveness. While Put/Get operations are performed by their callers, ARMCI uses one separate thread per process for progress of *accumulate* operations.

The implementation of collective operations has received its fair share of attention. Yang and Wang [32, 33] discussed algorithms for near optimal all-to-all broadcast on meshes and tori. Kumar and Kale [15] discussed algorithms to optimize all-to-all multicast on fat-tree networks. Thakur et al [26] discussed the scalability of MPI collectives and described implementations that use multiple algorithms in order to alleviate congestion in data intensive operations such as all-to-all. All these algorithms initiate non-blocking communication with a large number of peers, thus our approach can be transparently retrofitted on their implementations.

8. CONCLUSION

In this paper we have explored the design aspects of a dedicated parallel communication runtime that handles message injection and scheduling on behalf of application level tasks. Our runtime is able to increase the instantaneous communication concurrency and provide near saturation bandwidth, independent of the application configuration and its dynamic behavior.

We strive to provide performance and portability by: 1) using a dual "parallelization" strategy where tasks dedicated to communication are instantiated as either processes or `pthreads`; 2) using a selective parallelization strategy guided by network saturation performance models; and 3) implementing either cooperative scheduling or core partitioning schemes.

This architecture is well suited for hybrid parallelism implementations that combine intra- and inter-node programming models, as well as dynamic tasking programming models. We show very good performance improvements for collective operations, as well as hybrid parallelism codes. As HPC systems with many cores per chip are deployed, such as the 72-core Intel Knight's Landing, core partitions dedicated to communication become feasible. This alleviates the need for improving the load balancing and cooperative kernel level task scheduling mechanisms.

Unfortunately, if performance portability is a goal, a dual parallelization strategy seems to be required for the near to medium future. Furthermore, during this work we uncovered limitations in existing system software in the area of memory registration and job spawning. These unnecessarily complicate the implementation of multithreaded runtimes such as ours.

9. REFERENCES

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks – summary and preliminary results. In *Supercomputing*, pages 158–165, New York, NY, USA, 1991. ACM.

[2] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. L. Traeff. MPI on Millions of Cores. *Parallel Processing Letters (PPL)*, 21(1):45–60, Mar. 2011.

[3] F. Blagojević, P. Hargrove, C. Iancu, and K. Yelick. Hybrid PGAS runtime support for multicore nodes. In *Conference on Partitioned Global Address Space Programming Model*, PGAS '10, 2010.

[4] D. Bonachea. GASNet Specification, v1.1. Technical Report CSD-02-1207, University of California at Berkeley, October 2002.

[5] Berkeley UPC. http://upc.lbl.gov.

[6] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. In *Supercomputing*, pages 12–12, Nov 2000.

[7] P. Charles, C. Donawa, K. Ebcioglu, et al. X10: An object-oriented approach to non-unifrom cluster computing. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, October 2005.

[8] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan. Integrating Asynchronous Task Parallelism with MPI. *Parallel and Distributed Processing Symposium, International (IPDPS)*, 2013.

[9] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur. Enabling MPI interoperability through flexible communication endpoints. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 13–18, 2013.

[10] G. Dózsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur. Enabling concurrent multithreaded MPI communication on multicore petascale systems. In *European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, EuroMPI'10, 2010.

[11] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[12] D. Goodell, P. Balaji, D. Buntinas, G. Dozsa, W. Gropp, S. Kumar, B. R. d. Supinski, and R. Thakur. Minimizing MPI resource contention in multithreaded multicore environments. In *IEEE International Conference on Cluster Computing*, CLUSTER '10, 2010.

[13] Habanero-C. wiki.rice.edu/confluence/display/HABANERO/Habanero-C.

[14] K. Z. Ibrahim and K. A. Yelick. On the conditions for efficient interoperability with threads: an experience with PGAS languages using Cray communication domains. In *ICS*, 2014.

[15] S. Kumar and L. V. Kale. Scaling All-to-All Multicast on Fat-tree Networks. In *ICPADS'04*, page 205, 2004.

[16] M. Luo, X. Lu, K. Hamidouche, K. Kandalla, and D. K. Panda. Initial study of multi-endpoint runtime for mpi+openmp hybrid programming model on multi-core systems. *SIGPLAN Not.*, 49(8), Feb. 2014.

[17] M. Luo, D. K. Panda, K. Z. Ibrahim, and C. Iancu. Congestion Avoidance on Manycore High Performance Computing Systems. In *ICS*, 2012.

[18] K. Madduri, K. Z. Ibrahim, S. Williams, E.-J. Im, S. Ethier, J. Shalf, and L. Oliker. Gyrokinetic toroidal simulations on leading multi- and manycore HPC systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 23:1–23:12, 2011.

[19] K. Madduri, E.-J. Im, K. Z. Ibrahim, S. Williams, S. Ethier, and L. Oliker. Gyrokinetic particle-in-cell optimization on emerging multi- and manycore platforms. *Parallel Comput.*, 37(9):501–520, 2011.

[20] MPI Solutions for GPUs. https://developer.nvidia.com/mpi-solutions-gpus.

[21] miniGMG website. http://crd.lbl.gov/groups-depts/ftg/projects/current-projects/xtune/miniGMG.

[22] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy libray for ditributed array libraries and compiler run-time systems. In *IPPS/SPDP'99 Workshops*, 1999.

[23] R. Rabenseifner, G. Hager, and G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436, Feb 2009.

[24] M. Si, A. J. Peña, P. Balaji, M. Takagi, and Y. Ishikawa. MT-MPI: Multithreaded mpi for many-core environments. In *ICS*, pages 125–134, 2014.

[25] R. Thakur and W. Gropp. Test suite for evaluating performance of multithreaded MPI communication. *Parallel Comput.*, 35(12), 2009.

[26] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *IJHPCA*, pages 49–66, 2005.

[27] UPC Consortium. upc.lbl.gov/docs/user/upc_spec_1.2.pdf.

[28] UPC Consortium. UPC Optional Library Specifications- version 1.3. upc-specification.googlecode.com/files/upc-lib-optional-spec-1.3-draft-3.pdf, Nov. 2012.

[29] UPC-FT benchmark. https://www.nersc.gov/users/computational-systems/nersc-8-system-cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/npb-upc-ft/.

[30] S. Williams, D. D. Kalamkar, A. Singh, A. M. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker. Optimization of geometric multigrid for emerging multi- and manycore processors. In *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12. IEEE Computer Society Press, 2012.

[31] S. Williams, L. Oliker, J. Carter, and J. Shalf. Extracting ultra-scale lattice boltzmann performance via hierarchical and distributed auto-tuning. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11. ACM, 2011.

[32] Y. Yang and J. Wang. Efficient All-to-All Broadcast in All-Port Mesh and Torus Networks. In *International Symposium on High Performance Computer Architecture*, HPCA '99, pages 290–, Washington, DC, USA, 1999. IEEE Computer Society.

[33] Y. Yang and J. Wang. Near-Optimal All-to-All Broadcast in Multidimensional All-Port Meshes and Tori. *IEEE Trans. Parallel Distrib. Syst.*, 13:128–141, February 2002.