libMPNode: An OpenMP Runtime For Parallel Processing Across Incoherent Domains

Robert Lyerly
Bradley Department of Electrical and Computer Engineering
Virginia Tech
USA
rlyerly@vt.edu

Sang-Hoon Kim
Department of Software and Computer Engineering
Ajou University
South Korea
sanghoonkim@ajou.ac.kr

Binoy Ravindran
Bradley Department of Electrical and Computer Engineering
Virginia Tech
USA
binoy@vt.edu

Abstract

In this work we describe libMPNode, an OpenMP runtime designed for efficient multithreaded execution across systems composed of multiple non-cache-coherent domains. Rather than requiring extensive compiler-level transformations or building new programming model abstractions, libMPNode builds on recent works that allow developers to use a traditional shared-memory programming model to build applications that are migratable between incoherent domains. libMPNode handles migrating threads between domains, or *nodes*, and optimizes many OpenMP mechanisms to reduce cross-node communication. While applications may not scale as written, we describe early experiences in simple code refactoring techniques that help scale performance by only changing a handful of lines of code. We describe and evaluate the current implementation, report on experiences using the runtime, and describe future research directions for multi-domain OpenMP.

CCS Concepts • **Computing methodologies → Parallel computing methodologies**; • **Computer systems organization → Multicore architectures**;

Keywords OpenMP, non-cache-coherent multicores, scalability

ACM Reference Format:
Robert Lyerly, Sang-Hoon Kim, and Binoy Ravindran. 2019. libMPNode: An OpenMP Runtime For Parallel Processing Across Incoherent Domains. In *The 10th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'19), February 17, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3303084.3309495

PMAM'19 , February 17, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6290-0/19/02…$15.00
https://doi.org/10.1145/3303084.3309495

1 Introduction

In recent years there has been an increasing shift towards multicore, heterogeneous and distributed architectures to compensate for the limited increases in single core scalability [13, 27]. Additionally, there has been an explosion in big data – data volumes are projected to grow 40% every year for the next decade [30], meaning there is insatiable demand for compute power to mine new and increasing sources of data. Computer systems are undergoing a revolution to deal with this data, from the architecture to system software and applications.

Because of these trends and due to the complexities of scaling interconnects to larger core counts [17], the systems community has experienced new interest in moving from software architectures for fully cache-coherent systems to those composed of multiple incoherent domains, where cache coherence is provided only within a domain, as shown in Figure 1. Recent works have developed software architectures for mobile SoCs containing incoherent CPU cores [1, 19], servers with incoherent compute elements [3, 4, 12] or even rack-scale clusters [21–23]. In these architectures, the software treats each cache coherence domain as a *node* and provides cross-node execution and memory coherence transparently to applications. This allow developers to continue using traditional shared-memory programming models to target multi-domain systems, eliminating the need for new programming models [7, 9, 11, 22] or complex compiler techniques and runtime systems that are limited due to the visible split between cache coherence domains [18, 20, 31].

However, while these new software systems provide the ability to execute shared-memory applications across multiple domains, running multithreaded applications like those parallelized with OpenMP as written (i.e., distributing threads across all domains) can incur significant overheads due to cross-domain communication. In particular cross-domain data accesses caused by false sharing [29], synchronization or even regular memory accesses are orders of magnitude slower than DRAM accesses due to software-provided memory consistency. This problem is exacerbated with increasing numbers of domains, as the software must spend more time

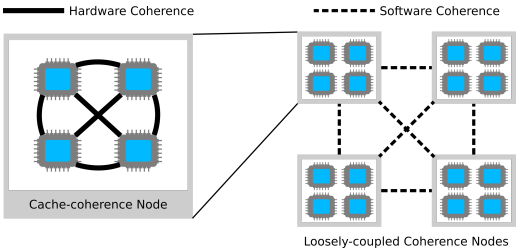

Figure 1. Due to the complexities of scaling hardware cache coherence (left), systems designers have begun coupling together multiple incoherent domains or *nodes* via interconnects like system buses or high-speed links. Hardware coherence is provided within a single node but not across nodes (right) – the software, i.e., OS, middleware or user application, handles cross-node coherence.

enforcing cross-domain consistency. *How can OpenMP applications, including the runtime and application itself, be optimized for cross-domain multithreaded execution?*

In this work, we describe the design and implementation of a new runtime named `libMPNode`, which allows developers to create multi-domain applications from existing OpenMP code by leveraging software-provided shared-memory abstractions. Because of flexibility of the shared-memory abstraction, we were able to survey applications from several benchmark suites without any code modifications. From this initial survey we identified several sources of bottlenecks, both in the OpenMP runtime and in how OpenMP was used by the developer. For the runtime, we optimized `libMPNode` to be domain-aware and reorganized OpenMP initialization and synchronization to minimize cross-domain communication. For OpenMP usage, we identified several OpenMP best-use practices to increase multi-domain scalability. Finally, we identified several future directions for multi-domain optimization. We present the following contributions:

- We describe `libMPNode`'s design and implementation, including parallel team creation, hierarchical primitives for efficient synchronization and compiler data placement optimizations;
- We present initial usage experiences, including how developers should utilize OpenMP's directives to remove unnecessary overheads;
- We present an initial evaluation of `libMPNode` on an emulated multi-domain system comprised of servers interconnected via high-speed network. Our results reveal that `libMPNode`'s multi-node implementation achieves up to a 38x speedup on 8 nodes for synchronization primitives versus a naïve implementation and provides a geometric mean speedup of 3.27x for scalable applications versus the fastest time on a single node;
- We identify remaining bottlenecks caused by data transfer overheads and propose future directions for better utilizing cross-domain interconnect bandwidth.

2 Related Work

Traditionally, developers have used the message passing interface (MPI) to distribute execution across nodes [11]. Deemed the "assembly language of parallel processing" [18], MPI forces developers to orchestrate parallel computation and manually keep memory consistent across nodes through low-level send/receive APIs, which leads to complex applications [4]. Partitioned global address space (PGAS) languages like Unified Parallel C [9] and X10 [7] provide language, compiler and runtime features for a shared memory-esque abstraction on clusters. How threads access global memory on remote nodes is specific to each language, but usually relies on a combination of compiler transformations/runtime APIs and requires the user to define thread and data affinities (i.e., which threads access what data). More recently, many works have re-examined distributed shared memory abstractions in the context of new high-bandwidth interconnects. Grappa [22] provides a PGAS programming model with many runtime optimizations to efficiently distribute computation across a cluster with high-speed interconnects. Grappa relies on a tasking abstraction to hide the high costs of remote memory accesses through massive parallelism, meaning many types of applications may not fit into their framework.

Previous works evaluate OpenMP on software distributed shared memory systems [5, 14, 20]. These approaches require complex compiler analyses (e.g., inter-procedural variable reachability) and transformations (software DSM consistency boilerplate, data privatization) in order to translate OpenMP to DSM abstractions, which limit their applicability. OpenMP-D [18] is another approach whereby the compiler converts OpenMP directives into MPI calls. This process requires sophisticated data-flow analyses and runtime profiling/adaptation to precisely determine data transfers between nodes. Additionally, OpenMP-D limits its scope to applications that repeat an identical set of computation multiple times. OmpCloud [31] spans OpenMP execution across cloud instances using OpenMP 4.5's offloading capabilities [24]. However, computation must fit into a map-reduce model and developers must manually keep memory coherent by specifying data movement between nodes.

Previous works have studied migrating threads between multiple nodes. Operating systems like Kerrighed [21], K2 [19] and Popcorn Linux [15, 16, 25] provide a *single system image* between non-cache-coherent nodes interconnected via high speed links. In particular, they provide the ability to *migrate* threads between nodes by transplanting the thread's context. Additionally, they provide a *distributed shared memory* (DSM) abstraction to applications which allow threads to run on multiple nodes as if they were cache coherent. The DSM layer migrates data on-demand between nodes by intercepting OS page faults and transferring page data using message passing between the kernel instances on each node.

```
int vecsum(const int *vec, size_t num) {
    size_t i;
    int sum = 0;
#pragma omp parallel for reduction(+:sum)
    for(i = 0; i < num; i++) sum += vec[i];
    return sum;
}
```

Listing 1. OpenMP vector addition

While there are variations, most DSM implementations use a multiple reader/single writer protocol which enforces sequential consistency across machines at a page granularity. These two abstractions are integrated deeply into the OS so that unmodified shared memory applications can run across multiple non-cache-coherent nodes. None of these works, however, have studied or optimized execution of a single application across multiple nodes and the performance implications of the shared-memory abstraction for simultaneous cross-node execution.

3 Design & Implementation

In this section we describe the design and implementation of `libMPNode` for transparently running multithreaded applications across multiple incoherent domains. `libMPNode` leverages Popcorn Linux [16], namely transparent thread migration and distributed shared memory, to allow application developers to use a well-studied and simple shared memory parallel programming model in multi-node systems. Cross-node thread migration is implemented similarly to OS continuations [15]; threads begin by making a system call into the thread migration service. The original thread is put to sleep on the source node, and a new thread is instantiated with the original thread's context on the remote node, returning to user-space to continue normal execution. Popcorn Linux provides an OS-level abstraction for distributed shared memory. By carefully managing page permissions, the OS can force threads accessing remote memory to fault, allowing the OS to transparently intercede on the thread's behalf. Data pages are migrated between nodes to optimize for locality, similarly to a cache coherence protocol. Popcorn Linux uses a *multiple-reader/single-writer* protocol [25], which enforces sequential consistency between nodes. Multiple nodes may have read-only copies of a data page (and may access it in parallel), but nodes must acquire exclusive access to a page in order to write to it. Nodes invalidate other copies of the page before they may gain exclusive access, preserving the single-writer invariant. Because the DSM is implemented transparently by the OS, existing shared-memory applications can execute across nodes unmodified. The complete details of Popcorn Linux can be found in past works [3, 4, 15, 16, 25].

`libMPNode` is node-aware and organizes execution so as to minimize cross-node communication by placing threads into a per-node hierarchy during team startup (Section 3.1). During subsequent execution, `libMPNode` breaks OpenMP synchronization primitives into local and global components (Section 3.2). Even though this design implements OpenMP abstractions more efficiently than a node-unaware runtime, the user can have a significant impact on performance. We discuss how several sources of inefficiency within applications can be refactored to further optimize execution (Section 3.3).

3.1 Distributed OpenMP Execution

OpenMP consists of a set of compiler directives and runtime APIs which control creating *teams* of threads to execute code in parallel. In particular, the developer annotates source code with OpenMP pragmas, i.e., `#pragma omp`, which direct the compiler to generate parallel code regions and runtime calls to the OpenMP runtime. Developers spawn teams of threads for parallel execution by adding `parallel` directives to structured blocks, which the compiler outlines and calls through the runtime. Additionally, OpenMP specifies pragmas for *work-sharing* between threads in a team (e.g., `for`, `task`) and synchronization primitives (e.g., `barrier`, `critical`) among other capabilities. Listing 1 shows an example of parallelizing vector sum with OpenMP. The `parallel` directive instructs the compiler and runtime to create a team of threads to execute the for-loop. The `for` directive instructs the runtime to divide the loop iterations among threads in the team. The `reduction` clause informs the runtime that threads should sum array elements into thread-local storage (i.e., the stack) and accumulate the value into the shared `sum` variable at the end of the work-sharing region. Finally, the `parallel` and `for` directives include an implicit ending barrier.

Internally, OpenMP functionality is implemented by a combination of compiler transformations and runtime calls. The compiler outlines parallel blocks into separate functions and inserts calls to a "parallel begin" API to both fork threads for the team and call the outlined function. Other directives are also implemented as API calls – a `for` directive is translated into a runtime call which determines the lower and upper bounds of the loop iteration range for each thread and synchronization primitives are implemented as calls into the runtime to wait at a barrier or execute a critical section. While the developer can very easily parallelize and synchronize team threads using these pragmas, their implementation can drastically affect performance. OpenMP assumes a homogeneous memory hierarchy, where accesses to global memory are relatively uniform from all compute elements in terms of latency. However for multi-node systems this assumption is broken and accesses to arbitrary global memory (e.g., reducing data or waiting at barriers) can cause severe performance degradations. In order to minimize cross-node traffic, `libMPNode` refactors the OpenMP runtime to break functionality down into *local* (intra-node) and *global* (inter-node) execution.

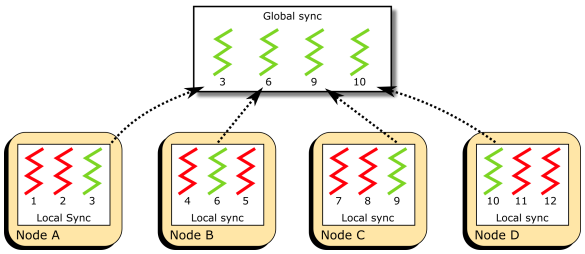

Figure 2. Thread synchronization hierarchy. Red threads perform local synchronization while green threads (node leaders, threads 3, 6, 9 and 10) globally synchronize before performing local synchronization. A hierarchy reduces the number of threads performing global synchronization and thus cross-node traffic.

Initializing thread teams. To begin a parallel region, the OpenMP runtime forks team threads which call the outlined parallel region to begin execution. During team startup, `libMPNode` creates a logical *thread hierarchy* to break OpenMP functionality into local and global computation. Threads operate on per-node data structures whenever possible to avoid cross-node data transfers. The first place this is utilized is when the runtime communicates parallel region startup information to threads. When the main thread starts a new parallel region, it must communicate both the outlined function and other execution state (references to shared variables, work sharing data structures, etc.) to all threads executing the new parallel region. Most OpenMP runtimes copy this information directly into each thread's thread local storage. However in a DSM-based system, this incurs 2 transfers for each thread – one for the main thread to write the startup data and one for each thread to read. Because this information is common to all threads in the team, `libMPNode` instead sets this data once per node and threads synchronize per-node to consume this information (see "Synchronizing Threads" below).

Migrating Team Threads. OpenMP provides facilities for specifying where threads should execute. In particular, OpenMP v4 [24] describes a method for mapping threads to physical "places" like sockets, cores and hardware threads. `libMPNode` extends this capability with a "`nodes`" keyword that allows users to transparently distribute threads across nodes, while internally initializing the thread hierarchy to match. `libMPNode` parses the places specification at application startup. Threads forked at the beginning of a parallel section enter a generic startup function inside `libMPNode` where the runtime applies the placement specification to migrate threads from the *origin* to *remote* nodes according to the user specification. `libMPNode` calls into the kernel's thread migration service to transparently migrate to new nodes. Threads execute as if they had never left the origin – data is brought over on-demand and kept consistent using the DSM layer. Post-migration, threads call into the outlined parallel region and execute as if on a single shared memory machine. At the end of execution, threads migrate back to the origin for cleanup. Thus, developers can distribute threads across nodes without changing a single line of code within the application – `libMPNode` encapsulates all the machinery necessary for interacting with the OS to migrate threads between nodes. This also gives the runtime flexibility to redistribute threads between nodes if needed; for example, to co-locate threads accessing the same memory.

Synchronizing threads. In order to facilitate optimizations listed in Section 3.2, `libMPNode` logically organizes threads into local/global hierarchy for synchronization. This enables optimizations that mitigate cross-node traffic, a source of major overheads in DSM systems (including [16]). `libMPNode` uses a *per-node leader selection* process whereby a leader is selected from all threads executing on a given node to participate in global synchronization (all other non-leader threads synchronize within a node). As illustrated in Figure 2, this allows `libMPNode` to reduce contention while providing the same semantics as a normal synchronization. `libMPNode` provides two types of selection processes depending on whether a *happens-before* ordering is required:

1. **Optimistic selection.** The first thread on a node to arrive at the synchronization point is selected as the node's leader. The leader executes global synchronization while other threads on the node continue in parallel, allowing all threads to perform useful work without blocking. After a global synchronization, leaders communicate results with local threads. This is useful for synchronization which does not require any ordering, e.g., `reduction` operations.
2. **Synchronous selection.** The last thread to arrive at the synchronization point is selected as the leader and the last per-node leader to arrive at the global synchronization point performs any global work required. This is useful for synchronization which requires a *happens-before* ordering, e.g., for `barriers` all threads must arrive at the synchronization point *before* any threads are released.

3.2 Optimizing OpenMP execution

By controlling thread distribution across nodes and organizing threads into a hierarchy, `libMPNode` can reduce several sources of cross-node overhead. First, accessing remote memory takes two orders of magnitude longer in [16] than local DRAM accesses, meaning `libMPNode` organizes as much computation as possible to be performed locally. Second, the DSM layer operates at a page granularity which can cause pages to "ping pong" when threads on multiple nodes access the same or discrete data on the same page. Logically organizing memory into per-node partitions can again yield large speedups. In this section we describe compiler and runtime optimizations to reduce these two sources of overhead.

Hierarchical Barriers. OpenMP makes extensive use of barriers for synchronization at the end of many directives.

```
struct barrier_t PAGE_ALIGN
{ int rem, total, sleep_key; };
struct hybrid_barrier_t
{ barrier_t local[NUM_NODES], global; };

void barrier_wait(hybrid_barrier_t *bar) {
    int thr_rem = atomic_sub(&bar->local.rem,1);
    if(thr_rem) {
        /* Spin a while before sleeping,
           return if all threads arrive */
        if(do_spin(bar)) return;
        /* Sleep until all threads arrive */
        else sleep(&bar->local.sleep_key);
    } else { /* Per-node leader */
        global_barrier(bar);
        bar->local.rem = bar->local.total;
        wake(&bar->local.sleep_key);
    }
}

void global_barrier(hybrid_barrier_t *bar) {
    int n_rem = atomic_sub(&bar->global.rem,1);
    /* Sleep until all leaders arrive */
    if(n_rem) sleep(&bar->sleep_key);
    else { /* Wake up other leaders */
        bar->rem = bar->total;
        wake(&bar->sleep_key);
    }
}
```

Listing 2. Hierarchical barrier pseudocode. Threads call into `barrier_wait()` and check to see if they are the last thread to arrive for the node. If not, they wait at the local barrier, only ever touching data already mapped to the node. If they are the last thread, they are elected the node's leader and synchronize at the global barrier.

For many OpenMP runtimes, barriers are implemented using a combination spin-sleep approach where threads spin until some condition becomes true, or sleep if it does not within some fixed interval. While suitable for shared memory systems where there are few threads and cache-line contention is relatively cheap, this form of synchronization causes enormous overheads in multi-node systems as many threads spin-wait on multiple nodes and cause the DSM layer to thrash. `libMPNode` avoids this by using hierarchical local/global barriers. A hierarchical barrier consists of a local spin-wait barrier for each node and one top-level global barrier as shown in Listing 2. Threads use a synchronous selection process to pick a per-node leader; all threads not selected wait at their respective local barriers. A synchronous selection process is required in order to establish a *happens-before* relationship between all threads arriving at the barrier on all nodes and the global barrier release. Otherwise, threads could be released on individual nodes before all threads executing on all nodes had reached the barrier. The per-node leaders wait at the global barrier for all nodes to arrive. Threads entering the global barrier, as shown in Listing 2, do not spin but instead do a single atomic operation, which reduces cross-node contention on the global barrier's state[1]. Once all leaders reach the global barrier, they are released and join the local barriers to release the rest of the threads. Note that all barriers, both local and global, are placed on separate pages to avoid cross-node contention.

Hierarchical reductions. Similarly, reductions can also be broken down into local and global computation. OpenMP requires that reductions are both associative and commutative [24], meaning they can be performed in any order and thus do not require a happens-before relationship. `libMPNode` uses an optimistic leader selection process to pick per-node leaders to reduce data for each node. The leader waits for threads to produce data for reducing, allowing threads to execute in parallel while it performs the reduction operation. Once the leader has reduced all data from its node, it makes the node's data available for the global leader (which is also selected optimistically). The global leader pulls data from each node for reduction, producing the final global result. The hierarchy again reduces cross-node traffic as reduction data is only transferred once per node.

Moving Shared Variables to Global Memory. OpenMP describes a number of *data-sharing attributes* which describe how threads executing parallel regions access variables in enclosing functions. Developers can specify variables as *private*, meaning all threads get their own copy of the variable, or *shared*, meaning all threads read and write the same instance of the variable. For shared variables, the compiler typically allocates stack space on the main thread's stack and passes a reference to this storage to all threads executing the parallel region. In a multi-node setting this leads to false sharing as threads reading/writing the shared variables contend with the main thread as it uses its stack for normal execution. To avoid this situation we modified `clang` to copy shared variables to global memory for the duration of the parallel region so that threads accessing these variables do not access the main thread's stack pages. Copying shared variables to and from global memory could cause high overheads in situations with many and/or large shared variables. However, we did not find this situation in the benchmarks we evaluated.

Future optimizations. Similar to the hierarchical barriers, other synchronization and work sharing primitives such as `critical` directives and dynamically scheduled work-sharing regions can benefit from a hierarchical thread organization using an optimistic leader selection process to reduce inter-node traffic. We leave these engineering optimizations as future work.

3.3 Using OpenMP Efficiently

When developing `libMPNode` we discovered several OpenMP usage patterns that cause sub-optimal behavior in multi-node

[1]Global synchronization could be further optimized with new kernel-level multi-node primitives

```
/* Sub-optimal - start many parallel regions */
for(j = 0; j < NUM_RUNS; j++) {
#pragma omp parallel for private(i, price, priceDelta)
    for(i = 0; i < numOptions; i++)
        ...(compute)...
}
/* Better - separate parallel and
   work-sharing directives */
#pragma omp parallel private(i, price, priceDelta)
for(j = 0; j < NUM_RUNS; j++) {
#pragma omp for
    for(i = 0; i < numOptions; i++)
        ...(compute)...
}
```

Listing 3. `blackscholes` parallel region optimization. Rather than starting many parallel regions, users should start fewer regions with multiple work sharing regions.

settings. Many of these sources of overhead can be rectified by small code modifications. Here we detail how developers can avoid these overheads.

Remove excessive parallel region begins. Each `parallel` directive causes the compiler to generate a new outlined function and the runtime to start a new thread team to execute the parallel region. While most OpenMP runtimes maintain a thread pool to avoid overheads of re-spawning threads, each encountered parallel region causes communication with the main thread, e.g., passing function and argument pointers to team threads in order to execute the region. Even with the previously described per-node team start optimization, this can cause high overheads for applications that start large numbers of parallel regions. As shown in Listing 3 for the `blackscholes` benchmark, users should lift parallel directives out of loops to avoid these initialization overheads wherever possible.

Access memory consistently across parallel regions. Cross-node execution overheads are dominated by the DSM layer, and thus data placement in the cluster. In order to minimize data movement, threads should use the same data access patterns when possible to avoid shuffling pages between nodes. Listing 4 shows an example from `cfd` where a copy operation accesses memory in a different pattern from the compute kernel. The optimized version (`copy_distributed()`) instead copies data using the same access pattern.

Use `master` instead of `single` directives. OpenMP provides a number of easy-to-use synchronization primitives, but users should substitute `single` for `master` directives when possible. `single` directives require two levels of synchronization – the first thread to encounter the single block executes the contained code while other threads skip the block and wait at an implicit barrier. This functionality is implemented by atomically checking if a thread is the first to arrive. However this synchronization operation requires cross-node traffic, leading to significant overheads. Users should utilize `master` and `barrier` directives together to implement the same semantics. The `master` directive specifies

```
void compute_step_factor(...) {
#pragma omp parallel for
    for(int blk=0; blk < nelr/blength; ++blk) {
        int b_start = blk * blength,
            b_end = (blk + 1) * blength;
        for(int i = b_start; i < b_end; i++) {
            float density = variables[i+...];
            ...(compute)...
        }
    }
}
/* Sub-optimal - does not access arrays in
   same way as compute_step_factor() */
void copy(...) {
#pragma omp parallel for
    for(int i = 0; i < N; i++)
        old_var[i] = variables[i];
}
/* Better - use same memory access pattern */
void copy_distributed(...) {
#pragma omp parallel for
    for(int blk=0; blk < nelr/blength; ++blk) {
        int b_start = blk * blength,
            b_end = (blk + 1) * blength;
        for(int i = b_start; i < b_end; i++) {
            old_var[i+...] = variables[i+...];
            ...(other copying)...
        }
}
```

Listing 4. `cfd` memory access optimization. Threads should access memory consistently across all parallel regions where possible.

that only the main thread should execute a code block and requires no synchronization (threads maintain their own IDs). Thus, users get the same functionality with less overhead.

3.4 Implementation

`libMPNode` extends and optimizes GNU's `libgomp` [10] v7.2, an OpenMP implementation packaged with `gcc`. For the optimizations that require compiler-level code generation changes, we modified `clang/LLVM` v3.7.1 due to its cleaner implementation versus `gcc`. However, `clang` emits OpenMP runtime calls to LLVM's `libiomp` [28], a complex cross-OS and cross-architecture OpenMP implementation with 3 times more lines of code versus `libgomp`. We opted for simplicity and added a small translation layer (~400 lines of code) to `libMPNode` to convert between the two. Note that `libMPNode`'s design is not tied to the choice of either compiler or OpenMP runtime – we chose this particular combination simply for ease of implementation. In addition to the runtime changes, we added a small memory allocation wrapper around `malloc` that organizes memory allocations into per-node regions. This allowed us to remove sources of false sharing, i.e., if threads on separate nodes allocate data on the same page they can cause a large amount of contention and unintentionally bottleneck execution.

4 Evaluation

In this section we evaluate libMPNode on a small cluster to emulate a multi-domain setup, including where libMPNode currently provides good performance and areas of improvement for future research. In particular we evaluate the following questions:

- How do the OpenMP runtime optimizations described in Sections 3.1 and 3.2 scale to multiple nodes? [4.1]
- How do applications perform both with and without the optimizations described in Section 3.3 when scaled to multiple nodes? [4.2]
- What types of applications currently run well when using libMPNode and what types could benefit from future optimizations? [4.3]

Experimental Setup. We evaluated libMPNode on a cluster of 8 Xeon servers, each of which contains 2 Intel Xeon Silver 4110 processors (max 2.1GHz clock) and 96 GB of DDR4-2667MHz RAM. Each Xeon processor has 8 cores with 2-way hyperthreading for a total of 16 threads per processor, or 32 threads per server. We ran up to 16 threads per server due to scalability limitations in [16]. Each server is equipped with a Mellanox ConnextX-4 Infiniband adapter supporting bandwidth up to 56 Gbps.

Applications. We evaluated OpenMP benchmarks from PARSEC [6], Rodinia [8] and NASA Parallel Benchmark [2, 26] suites. We selected a subset of benchmarks that 1) had enough parallel work to scale across multiple nodes and 2) had representative performance characteristics from which we could draw conclusions about libMPNode's effectiveness. Of note, we were able to survey cross-node performance for 26 benchmarks by only re-compiling and re-linking with our infrastructure. We performed an initial evaluation of benchmarks to determine scalability and contention points, then optimized applications as described in Section 3.3. We consider execution on a single server as the baseline, as comparing to other cluster programming solutions would require either significant application refactoring or complex compiler/runtime extensions to support the applications (one of the major benefits of libMPNode).

4.1 Microbenchmarks

First we evaluated the effectiveness of the hierarchy in scaling multi-node synchronization for several OpenMP primitives. The first microbenchmark we ran spawns 16 threads on each node from 1 to 8 nodes (no cross-node execution for 1 node) and executes 5000 barriers in a loop. Figure 3 shows the average barrier latency with and without hierarchy optimizations. Clearly libMPNode's hierarchy provides much better scalability – a naïve implementation where all threads on all nodes wait at a single global barrier leads to tens of millisecond latencies that drastically increase with node count (up to 72.4 milliseconds for 8 nodes). Meanwhile

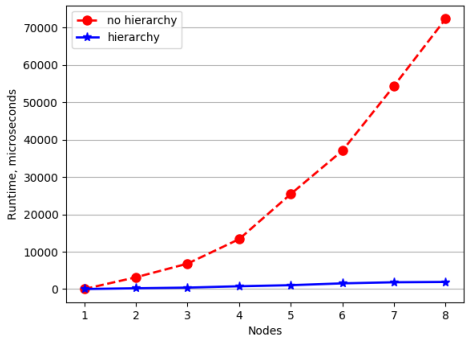

Figure 3. Barrier latency for different numbers of nodes with and without hierarchical barriers, in microseconds.

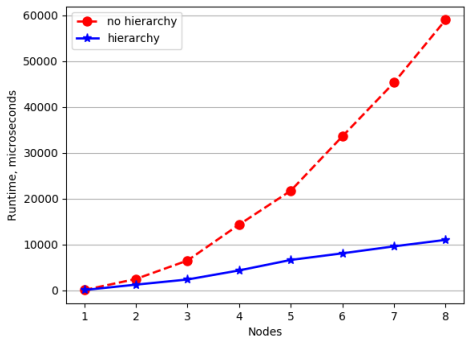

Figure 4. Reduction latency for different numbers of nodes with and without hierarchical reductions, in microseconds.

the hierarchical barrier leads to much better scalability with up to a 1.9 millisecond latency for 8 nodes, a 38x speedup.

We next ran a microbenchmark that sums all the elements in an array to stress cross-node parallel reductions. We again spawned 16 threads per node and allocated 50 pages of data for each thread to accumulate – each thread received the same amount of work to remove load imbalance effects on reduction latencies. Figure 4 shows the latency when performing a naïve reduction (all threads use atomic operations on a global counter) versus a hierarchical reduction (leaders first reduce locally and then globally). Similarly to barriers, hierarchical reductions have much better scalability than normal global reductions, taking 11 and 58 milliseconds on 8 nodes, respectively. Interestingly, the performance gap on 8 nodes between the normal and hierarchical reductions is only 5.4x. This is due to how the compiler implements reductions – the compiler allocates a thread-local copy of data to be reduced on each thread's stack and passes a pointer to that data to the runtime. Each per-node leader passes that pointer to the global leader for reduction, which causes contention on the per-node leader's stack page (global leader

reads reduction data, per-node leader uses stack for normal execution). Nevertheless, the hierarchy provides large performance benefits.

4.2 Benchmark Performance

Next, we ran benchmarks to evaluate the effectiveness of optimizations listed in Section 3.3. All benchmarks were run with hierarchical barriers and reductions enabled. Figures 5-10 show application performance when run with varying numbers of nodes (x-axis) and threads per node (trend lines). The y-axis shows runtime of each configuration in seconds; lower numbers mean better performance. Dotted red lines indicate application performance before optimization while solid blue lines indicate running time after optimization.

Application performance falls into three categories: applications that scale with more nodes (`blackscholes`, `EP`, `kmeans` and `lavaMD`), applications that exhibit some scalability but have non-trivial cross-node communication (`CG`) and applications that do not scale (`cfd`). For applications that scale, running with the highest thread count on 8 nodes led to a geometric mean speedup of 3.27x versus the fastest time on a single machine, or 4.04x not including `blackscholes` which has a significant sequential region. For CG, the fastest multi-node configuration achieved a slowdown of 5.4%, meaning there is plenty of room for performance optimization. The optimizations described in Sections 3.3 help for every multi-node configuration in every single application, although its effects are limited in those that do not scale. The scalable applications experience further performance gains by lifting shared variables into global memory and applying light-weight code modifications. `lavaMD` and `kmeans` experienced the largest benefits from optimizations, in particular lifting shared variables into global memory and using per-node memory allocations.

4.3 Performance Characterization

Here we describe application characteristics that have a significant impact on performance and future directions for further optimization in [16] and `libMPNode`.

Scalable applications. These applications have little-to-no communication between threads on different nodes and thus once the initial data exchange between nodes has been completed, threads run at full speed without inter-node communication. This is the ideal scaling scenario, but requires problems with large datasets that can be processed completely independently. Scalability is only limited by benchmark data size (`EP-C`, `kmeans`, `lavaMD`) or serial portions within the application (`blackscholes`).

Mildly-scaling applications. These applications share non-trivial amounts of data between threads during execution. For example, `CG-C` uses arrays of pointers to access sparse matrices through indirection, meaning there is little data locality when accessing matrix elements. We believe that prefetching up-to-date copies of data across nodes could significantly improve performance for these types of applications.

Non-scalable applications. These applications have many small parallel regions, low compute-to-memory ratios and continually shuffle pages between nodes. `cfd` iteratively scans one variables array and locally writes a new one. In the next iteration, these two array are swapped, leading to huge DSM layer overheads as writes must be propagated among nodes and reads that were replicated across nodes must be invalidated. There is not enough computation to amortize the cost of shuffling data. Instead, application developers would need to find alternate sources of parallelism, i.e., performing several of the computations in parallel. This could be achieved through nested OpenMP parallel regions; we leave implementing this functionality within the thread hierarchy as future work.

From the previously described characteristics, the main performance limitation was attributed to cross-node data shuffling in and between work-sharing regions. In order to further investigate system bottlenecks, we evaluated how much network bandwidth the DSM layer was able to utilize by running `cfd` on 2 nodes with 32 threads and capturing the number of pages transmitted in one second intervals to determine time varying bandwidth usage. Throughout the parallel portion of the application the messaging layer used on average 85.2 MB/s of bandwidth, close to two orders of magnitude less than 56 Gbps Infiniband can provide. This leads us to believe that future efforts should focus on how to use the ample available cross-node network bandwidth in order to better hide cross-node memory access latencies.

5 Future Work

There are numerous opportunities for future research with `libMPNode`. As previously mentioned, cross-node memory access latencies cause severe overheads for applications that shuffle large amounts of data between nodes. This can be attributed to two main factors: 1) distributed shared memory consistency overheads, and 2) on-demand data migration. In terms of DSM overheads, enforcing sequential consistency across nodes can block memory operations even when there is no data transfer involved. For example, in order to write to a page on a node, [16]'s DSM protocol first must invalidate permissions on all other nodes and then acquire write permissions (along with page data). Even when the node has the most recent page data (for example, after reading the page), a node must first invalidate permissions on other nodes before allowing threads write access due to sequential consistency semantics. However, OpenMP uses a release consistency model [24], meaning that [16]'s protocol provides stricter guarantees than is necessary for correct OpenMP semantics. Relaxing the DSM layer's consistency would eliminate much of the memory consistency maintenance overheads.

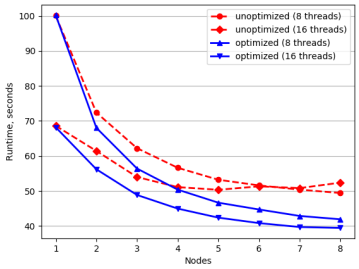

Figure 5. `blackscholes`

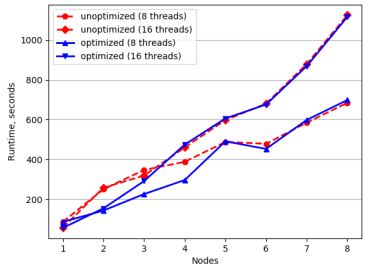

Figure 6. `cfd`

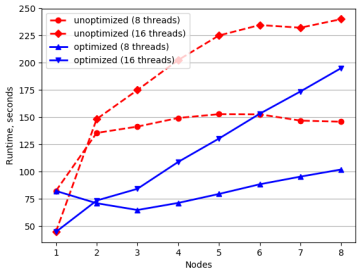

Figure 7. `CG` class C

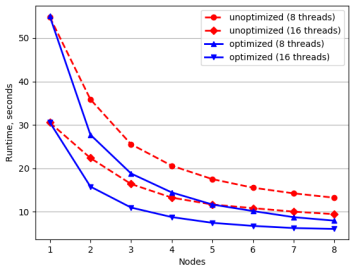

Figure 8. `EP` class C

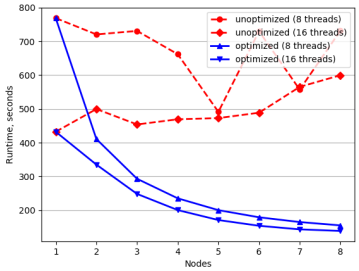

Figure 9. `kmeans`

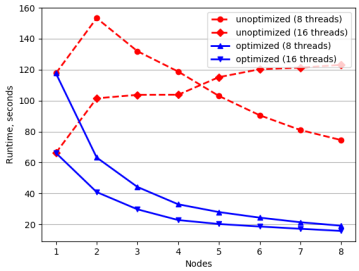

Figure 10. `lavaMD`

The second source of latency is due to the use of on-demand data migration. The DSM implementation observes memory accesses through the page fault handler and migrates data at the last possible moment. While this avoids migrating unused data, it places the data transfer latency directly in the critical path of execution. As mentioned previously, the interconnect provides ample unused cross-node bandwidth; techniques which leverage this bandwidth to preemptively place data can better hide cross-node latencies. For example, the compiler could place data "push" hints that inform the DSM when a thread has finished writing a page so that it can be proactively pushed to other nodes (similar in spirit to prefetching). Because OpenMP work sharing regions often structure memory accesses affine to loop iterations, the compiler could analyze memory access patterns in work sharing regions and inject data placement hints into the application.

6 Conclusion

In this work we described `libMPNode`, a runtime for easily creating multi-node applications using OpenMP. We described how `libMPNode` distributes application threads and provides a hierarchical thread organization for incoherent domain systems. Using this hierarchy, `libMPNode` optimizes synchronization including hierarchical barriers and reductions to speed up cross-node operations. Additionally, `libMPNode` includes compiler optimizations to lift shared variables to global memory to avoid cross-node false sharing. Using `libMPNode`, we detailed several OpenMP usage patterns that can be easily changed to achieve much better performance over multiple nodes. Finally, we evaluated `libMPNode` on a small cluster, demonstrating a 3.27x geometric mean speedup compared to single node execution. We also identified several limiting factors in both the application and infrastructure that we believe can be remedied further increase `libMPNode`'s performance. Because of `libMPNode`'s performance, we believe OpenMP is a viable and easy to use general purpose parallel programming model that can utilized for targeting emerging multi-node systems.

7 Acknowledgments

This work is supported in part by grants received by Virginia Tech including that from ONR under grant N00014-16-1-2711 and NAVSEA/NEEC under grant N00174-16-C-0018. Dr. Kim's work at Virginia Tech (former affiliation) was supported in part by ONR under grant N00014-16-1-2711, and his work at Ajou University was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2018R1C1B5085902).

References

[1] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. 2016. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 189–203. https://doi.org/10.1145/2872362.2872371

[2] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS parallel benchmarks. *The International Journal of Supercomputing Applications* 5, 3 (1991), 63–73.

[3] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. 2017. Breaking the Boundaries in Heterogeneous-ISA Datacenters. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 645–659. https://doi.org/10.1145/3037697.3037738

[4] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. 2015. Popcorn: Bridging the Programmability Gap in heterogeneous-ISA Platforms. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, Article 29, 16 pages. https://doi.org/10.1145/2741948.2741962

[5] A. Basumallik, S. J. Min, and R. Eigenmann. 2007. Programming Distributed Memory Sytems Using OpenMP. In *2007 IEEE International Parallel and Distributed Processing Symposium*. 1–8. https://doi.org/10.1109/IPDPS.2007.370397

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[7] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 519–538. https://doi.org/10.1145/1094811.1094852

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[9] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. 2005. An Evaluation of Global Address Space Languages: Co-array Fortran and Unified Parallel C. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05)*. ACM, New York, NY, USA, 36–47. https://doi.org/10.1145/1065944.1065950

[10] GNU. 2017. *GNU Offloading and Multi Processing Runtime Library: The GNU OpenMP and OpenACC Implementation*. Technical Report. https://gcc.gnu.org/onlinedocs/libgomp.pdf.

[11] William Gropp, Ewing Lusk, and Anthony Skjellum. 1999. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press.

[12] Charles Gruenwald, III, Filippo Sironi, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Hare: A File System for Non-cache-coherent Multicores. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, Article 30, 16 pages. https://doi.org/10.1145/2741948.2741959

[13] Tim Harris. 2015. Hardware Trends: Challenges and Opportunities in Distributed Computing. *SIGACT News* 46, 2 (June 2015), 89–95. https://doi.org/10.1145/2789149.2789165

[14] Jay P Hoeflinger. 2006. Extending OpenMP to clusters. *White Paper, Intel Corporation* (2006).

[15] D. Katz, A. Barbalace, S. Ansary, A. Ravichandran, and B. Ravindran. 2015. Thread Migration in a Replicated-Kernel OS. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. 278–287. https://doi.org/10.1109/ICDCS.2015.36

[16] Sang-Hoon Kim, Robert Lyerly, and Pierre Olivier. 2017. Popcorn Linux: Compiler, Operating System and Virtualization Support for Application/Thread Migration in Heterogeneous ISA Environments. Presented at the 2017 Linux Plumbers Conference. http://www.linuxplumbersconf.org/2017/ocw/proposals/4719.html.

[17] Akhilesh Kumar. 2017. Intel's New Mesh Architecture: The 'Superhighway' of the Data Center. (June 2017). https://itpeernetwork.intel.com/intel-mesh-architecture-data-center/.

[18] Okwan Kwon, Fahed Jubair, Rudolf Eigenmann, and Samuel Midkiff. 2012. A Hybrid Approach of OpenMP for Clusters. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 75–84. https://doi.org/10.1145/2145816.2145827

[19] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. 2014. K2: A Mobile Operating System for Heterogeneous Coherence Domains. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 285–300. https://doi.org/10.1145/2541940.2541975

[20] Seung-Jai Min, Ayon Basumallik, and Rudolf Eigenmann. 2003. Optimizing OpenMP Programs on Software Distributed Shared Memory Systems. *Int. J. Parallel Program.* 31, 3 (June 2003), 225–249. https://doi.org/10.1023/A:1023090719310

[21] C. Morin, R. Lottiaux, G. Vallee, P. Gallard, D. Margery, J. . Berthou, and I. D. Scherson. 2004. Kerrighed and data parallelism: cluster computing on single system image operating systems. In *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935)*. 277–286. https://doi.org/10.1109/CLUSTR.2004.1392625

[22] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-Tolerant Software Distributed Shared Memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 291–305. https://www.usenix.org/conference/atc15/technical-session/presentation/nelson

[23] Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. 2017. OS Support for Thread Migration and Distribution in the Fully Heterogeneous Datacenter. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*. ACM, New York, NY, USA, 174–179. https://doi.org/10.1145/3102980.3103009

[24] OpenMP Architecture Review Board. 2015. *OpenMP Application Program Interface v4.5*. Technical Report. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[25] Marina Sadini, Antonio Barbalace, Binoy Ravindran, and Francesco Quaglia. 2013. A page coherency protocol for popcorn replicated-kernel operating system. In *Proceedings of the ManyCore Architecture Research Community Symposium (MARC)*.

[26] S. Seo, G. Jo, and J. Lee. 2011. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*. 137–148. https://doi.org/10.1109/IISWC.2011.6114174

[27] Herb Sutter. 2012. Welcome to the Jungle. https://herbsutter.com/welcome-to-the-jungle/.

[28] The LLVM Project. 2015. *LLVM OpenMP Runtime Library*. Technical Report. http://openmp.llvm.org/Reference.pdf.

[29] J. Torrellas, H. S. Lam, and J. L. Hennessy. 1994. False sharing and spatial locality in multiprocessor caches. *IEEE Trans. Comput.* 43, 6 (June 1994), 651–663. https://doi.org/10.1109/12.286299

[30] Vernon Turner, John F Gantz, David Reinsel, and Stephen Minton. 2014. The digital universe of opportunities: Rich data and the increasing value of the internet of things. *IDC Analyze the Future* (2014), 5.

[31] Hervé Yviquel, Lauro Cruz, and Guido Araujo. 2018. Cluster Programming Using the OpenMP Accelerator Model. *ACM Trans. Archit. Code Optim.* 15, 3, Article 35 (Aug. 2018), 23 pages. https://doi.org/10.1145/3226112