An OpenMP Runtime for Transparent Work Sharing Across Cache-Incoherent Heterogeneous Nodes

Robert Lyerly
Virginia Tech
rlyerly@vt.edu

Changwoo Min
Virginia Tech
changwoo@vt.edu

Christopher J. Rossbach
University of Texas at Austin and VMware Research
rossbach@cs.utexas.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

Abstract

In this work we present `libHetMP`, an OpenMP runtime for automatically and transparently distributing parallel computation across heterogeneous nodes. `libHetMP` targets platforms comprising CPUs with different instruction set architectures (ISA) coupled by a high-speed memory interconnect, where cross-ISA binary incompatibility and non-coherent caches require application data be marshaled to be shared across CPUs. Because of this, work distribution decisions must take into account both relative compute performance of asymmetric CPUs and communication overheads. `libHetMP` drives workload distribution decisions without programmer intervention by measuring performance characteristics during cross-node execution. A novel HetProbe loop iteration scheduler decides if cross-node execution is beneficial, and either distributes work according to the relative performance of CPUs when it is, or places all work on the set of homogeneous CPUs providing the best performance when it is not. We evaluate `libHetMP` using compute kernels from several OpenMP benchmark suites and show a geometric mean 41% speedup in execution time across asymmetric CPUs.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; *Heterogeneous (hybrid) systems*; **Multicore architectures**; *Heterogeneous (hybrid) systems*; • **Software and its engineering** → *Distributed memory*; *Distributed memory*.

Keywords: heterogeneous-ISA CPUs, OpenMP, work sharing

Middleware '20, December 7–11, 2020, Delft, Netherlands

ACM ISBN 978-1-4503-8153-6/20/12…$15.00
https://doi.org/10.1145/3423211.3425679

ACM Reference Format:
Robert Lyerly, Changwoo Min, Christopher J. Rossbach, and Binoy Ravindran. 2020. An OpenMP Runtime for Transparent Work Sharing Across Cache-Incoherent Heterogeneous Nodes. In *21st International Middleware Conference (Middleware '20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3423211.3425679

1 Introduction

In recent years there has been a shift towards increasingly heterogeneous platforms in order to cope with the slowdown of Moore's Law [53]. As chip designers have faced resistance in scaling single-core [52] and multicore [20] performance due to physical limitations, they have responded by incorporating more specialized processors into systems [45, 46]. These emerging heterogeneous systems are increasingly necessary to deal with future challenges, e.g., Amazon has begun offering cloud instances with different types of CPUs to match analytics workloads [9] and the Summit supercomputer combines CPUs and GPUs for enhanced performance and power efficiency [2]. The path forward for tackling these challenges is through increasing architectural diversity.

Chip manufacturers have begun diversifying server-grade CPU designs to strike different levels of single-threaded performance, parallelism and energy efficiency. For example, Intel Xeon [33] CPUs package tens of cores with high single-threaded performance, whereas Cavium ThunderX2 [44] CPUs instead package a large number of lower-performance cores. At the same time, chip designers have begun tightly coupling heterogeneous compute elements for power and performance benefits – for example, Intel's Agilex platform combines Xeon CPUs with FPGAs and ARM processors into a single physical processor package [5]. These trends suggest future platforms may provide greater architectural diversity by integrating asymmetric general-purpose server-grade CPUs into a single motherboard or package.

We envision a system with heterogeneous CPUs, each of which has its own physical memory, connected by either a point-to-point connection such as PCIe [1] or a fast memory bus such as AMD's Infinity Fabric [34]. These CPUs will also likely use heterogeneous instruction set architectures (ISAs),

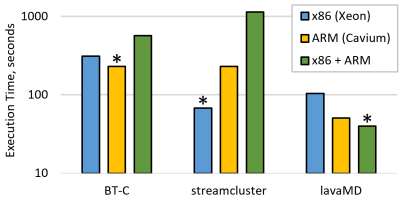


Figure 1. Execution time of OpenMP benchmarks with work-sharing regions executed entirely on an x86 Intel Xeon, entirely on an ARM Cavium ThunderX, and when leveraging both with `libHetMP`. *How can the runtime automatically determine the best workload distribution configuration across heterogeneous CPUs to optimize performance?*

which have been shown to provide better performance and energy efficiency than asymmetric single-ISA CPUs [19, 54]. For developers, the ability to optimize performance and energy efficiency of applications on such systems will be essential, but leveraging diverse architectures together is a daunting task. While cores within a homogeneous set of CPUs, termed a *node*, provide cache coherence, there is no coherence between heterogeneous CPUs. This necessitates software memory consistency and data movement between discrete memory regions. Additionally, because these CPUs use different ISAs, data must be *marshaled* to be shared. Data must either be laid out in a common format by the compiler or dynamically transformed when transferred at runtime. Traditional programming models like MPI [23] are a poor fit for programming these tightly-coupled heterogeneous servers as they require developers to manually manage memory consistency and work distribution.

Recent works such as K2 [35] and Popcorn Linux [7, 8, 30] instead use distributed shared memory (DSM) on tightly-coupled heterogeneous CPU systems for better programmability, transparency and flexibility. In addition to supporting cross-ISA execution migration [7, 19, 54], these systems provide *transparent* and *on-demand* data marshaling between nodes. Because of this transparency, multiple discrete memory regions appear as shared memory to applications. Distributing parallel computation across heterogeneous-ISA CPUs becomes simpler as parallel runtimes can assign work items to CPUs and let the DSM transparently marshal data. On-demand marshaling is expensive, however, and can have a significant performance impact in cross-node execution.

For tightly-coupled heterogeneous CPU platforms, the challenge is how to optimally distribute parallel work to balance per-CPU performance against DSM communication overheads. For example, Figure 1 shows the execution time of three OpenMP HPC benchmarks when run on an Intel Xeon E5-2620v4, a Cavium ThunderX, and when utilizing both simultaneously with the `libHetMP` runtime presented in this work. Because of cross-node traffic, it is not always beneficial for a parallel computation to utilize the processing resources of both CPUs together – BT-C is fastest when run entirely on the ThunderX's large number of small cores and streamcluster is instead fastest when run entirely on the Xeon. For lavaMD, however, utilizing both processors in tandem leads to the best performance as cross-node data accesses (and thus DSM traffic) are limited. Because no work distribution is best for all applications, *how can parallel workloads automatically and transparently leverage heterogeneous CPUs to maximize performance*?

We present `libHetMP`, an OpenMP runtime for automatically work sharing parallel computation in heterogeneous CPU systems. Our target platform consists of architecturally-diverse CPUs coupled via point-to-point connection or high speed bus. Because no such platform is commercially available, we emulate such a system by connecting servers with heterogeneous-ISA CPUs using Infiniband. `libHetMP` transparently reorganizes OpenMP execution for multi-node systems, eliminating sources of DSM overheads and extending existing OpenMP primitives to support heterogeneous CPUs. For work distribution, `libHetMP` uses a measurement-based approach to characterize an application's performance and automatically distribute work to nodes to achieve the best performance. `libHetMP` allows developers to use OpenMP, a well-known and mature parallel programming model, in emerging heterogeneous-ISA CPU systems while also abstracting away the underlying system architecture so developers do not have to reconfigure work distribution for these emerging systems. Although the current implementation targets 2-node systems, it is straightforward to extend `libHetMP` to systems with arbitrary numbers of nodes.

`libHetMP` includes a new *HetProbe* loop iteration scheduler that dynamically measures and distributes parallel work to best utilize heterogeneous CPUs. The HetProbe scheduler targets regular work-shared loops, where each loop iteration performs the same amount of computation using similar memory access patterns. The HetProbe scheduler analyzes the behavior of a small number of initial loop iterations and determines whether the remaining iterations should be executed across multiple nodes. If cross-node execution is beneficial, the HetProbe scheduler distributes work to threads based on the relative performance of each CPU. If work sharing across nodes causes too much communication, the HetProbe scheduler executes all work on the set of cache-coherent homogeneous CPUs best suited for a given computation. The difficult process of where to distribute work is transparently automated by `libHetMP`.

In this work we make the following contributions:

- The design and implementation of `libHetMP`, a new OpenMP runtime that distributes threads and parallel work across cache-incoherent heterogeneous CPU systems without programmer intervention;
- Extensions to shared memory OpenMP synchronization primitives and loop iteration schedulers to adapt execution to heterogeneous CPUs and minimize DSM overheads;

- Measurement tools built into the runtime that monitor metrics such as data transfer costs and hardware performance counters to make workload distribution decisions;
- The HetProbe loop iteration scheduler, which uses these metrics to automatically determine where to place computation in order to minimize DSM overheads and leverage architectural diversity to achieve the best performance;
- An evaluation of `libHetMP` using 10 benchmarks from 3 benchmark suites that shows up to a 4.7x speedup when work sharing across a Xeon and ThunderX versus homogeneous execution on the Xeon. We also show the HetProbe scheduler is able to make the right workload distribution choice in all benchmarks, including evaluating decisions on two interconnects.

2 Background

`libHetMP` is a multithreading runtime for OpenMP [42], a directive-based parallel programming model for C, C++ and Fortran applications. OpenMP is widely used in high-performance computing [24][18][48] because of its flexibility to express many different forms of parallel computation such as data parallelism (including hardware-based single instruction/multiple data computation), bag-of-tasks parallelism, synchronization and reductions [42]. OpenMP targets shared-memory systems (unlike programming models such as MPI [23] which uses message passing or OpenCL [28] for accelerators) but has continually evolved into new contexts, such as providing offloading directives for GPUs [42].

OpenMP specifies a set of directives that developers add to applications to parallelize execution. The compiler is responsible for converting OpenMP directives into function calls into the OpenMP runtime, which spawns teams of threads, partitions parallel work between threads and provides synchronization capabilities. For loop work sharing regions as shown in Listing 1 (e.g., `pragma omp for`), parallel work is distributed by assigning loop iterations to threads. The vector sum in Listing 1 shows an example of a work sharing region with regular loop iterations – each iteration performs the same amount of work and iterations have the same memory access patterns (affine accesses based on the loop iteration variable). Regular work-shared loops are a common class of data parallelism – with predictable compute and memory access patterns, they are amenable to fine-grained analysis and partitioning across multiple devices [36][49].

OpenMP assumes architectural uniformity and current implementations do not target heterogeneous-ISA CPUs. In order to support execution across such CPUs, the system software (compiler, OS, runtime) must provide a shared-memory abstraction. Even if this abstraction exists, optimizing OpenMP for heterogeneous CPU systems requires re-designing how parallel work is assigned to CPUs in consideration of system and interconnect performance characteristics. Before describing `libHetMP`, we first describe how previous works enable execution across heterogeneous-ISA

```
1  int vecsum(const int *vec, size_t num) {
2      size_t i;
3      int sum = 0;
4  #pragma omp parallel for reduction(+:sum)
5      for(i = 0; i < num; i++) sum += vec[i];
6      return sum;
7  }
```

Listing 1. OpenMP vector sum. OpenMP directives instruct the runtime to spawn threads, distribute loop iterations to threads and combine results from each thread.

CPUs. Throughout the work we refer to *nodes* as a set of single-ISA cache-coherent processors, e.g., each of the Xeon and ThunderX CPUs is considered a node.

Heterogeneous-ISA Execution. Unlike ARM's big.LITTLE architecture [21], which provides cache-coherence across same-ISA heterogeneous cores, there exist no server-grade cache-coherent heterogeneous CPUs. Past systems that couple together overlapping-ISA architectures (e.g., Xeon/Xeon Phi) are defunct; system designers wishing to couple together asymmetric processors today must integrate CPUs of different ISAs. Thus the system software (compiler, operating system, runtime) must handle both ISA heterogeneity and memory consistency. Previous works [7, 19, 54] describe system software for migrating compiled shared-memory applications between heterogeneous-ISA CPUs at runtime. While these works describe similar designs, we leverage Popcorn Linux [7] due to its availability.

Multi-ISA Binaries. Similarly to past works [19, 54], Popcorn Linux's compiler builds multi-ISA binaries which are capable of cross-ISA execution. Multi-ISA binaries consist of one aligned data section and multiple per-ISA code sections, one for each target ISA in the system. To enable cross-ISA execution, the compiler arranges the application's global address space to be aligned across ISAs so that pointers to globally-visible data and functions refer to the same addresses on all nodes. Additionally, the compiler generates metadata describing function stack layouts at *equivalence points* [55]. This metadata describes the locations and type information (e.g., pointer-type) of live values so that stack frames can be reconstructed for the destination ISA.

Thread Migration. Threads migrate between nodes at *migration points*, a subset of equivalence points chosen by the compiler or user. `libHetMP` adds migration points inside the OpenMP runtime to automate distributing thread teams across nodes. To migrate between nodes, threads enter a state transformation runtime that walks the thread's stack and transforms it to the destination ISA's layout. After transformation, threads pass a transformed register set to a migration system call and are returned to normal execution on the destination CPU with the registers (several works implement this mechanism for homogeneous-ISA [35, 38] or heterogeneous-ISA [7, 8, 30] systems). Unlike offloading

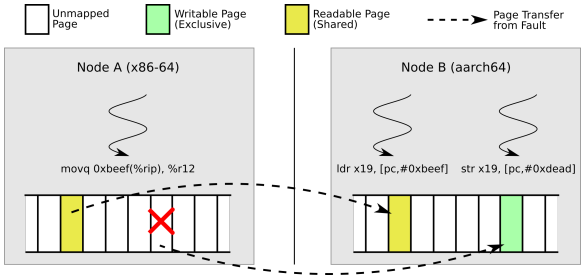


Figure 2. DSM protocol. Pages are migrated on-demand by observing memory accesses through the page fault handler. Pages read by threads on multiple nodes are *replicated* with read-only protections while only one node may have *exclusive* write permissions for a page.

where only a statically selected region of computation is executed on the target, transforming the stack allows threads to stay on target nodes for arbitrary lengths of time.

Page-level Distributed Shared Memory. Once threads have migrated to new nodes, they must be able to access application data. OS-level DSMs such as those proposed by Kerrighed [38], K2 [35] and Popcorn Linux observe remote memory accesses inside the page fault handler and migrate data pages similarly to a cache coherence protocol. By carefully manipulating page permissions, the OS forces the application to fault when accessing remote data. When a fault occurs, the kernel on the source (i.e., faulting) node requests the page from the remote node that currently owns the page. The page is transferred from the remote to the source and mapped into the application's address space. The memory access is restarted and application threads are unaware that data was fetched over the interconnect. In this way, data is marshaled between nodes transparently and on demand. Note that software memory consistency would be required even for heterogeneous CPUs with (cache-incoherent) shared memory in order to prevent lost or reordered writes due to differing memory consistency models.

Many DSM systems use a *multiple-reader, single-writer* protocol as shown in Figure 2. In addition to data, nodes request access rights based on the type of memory access. If multiple nodes read data from the same page, the protocol replicates the page with read-only permissions and all nodes can read the data in parallel. If a thread writes to a page, the node first invalidates all other copies of the page from other nodes and then acquires exclusive write access. Any subsequent attempts to read or write the page on other nodes will cause a fault and access rights must be re-acquired.

Cross-node Execution Challenges. Unlike traditional shared memory multiprocessor systems that share data at a cache-line granularity, DSM systems share data at a page granularity due to observing memory accesses via page faults. Additionally, the cost of bringing data over the interconnect and managing access permissions is significantly higher than a traditional memory access – rather than taking tens to hundreds of nanoseconds, page migration takes tens of microseconds (see Section 3). These two characteristics mean that in order for a parallel computation to benefit from leveraging multiple heterogeneous CPUs simultaneously, data accessed by threads on different nodes must partition cleanly between pages and there must be enough computation to amortize DSM costs. Otherwise, the application should only execute parallel work on a single node.

3 Design

`libHetMP` builds on Popcorn Linux's ability to distribute threads and transparently marshal data across heterogeneous-ISA CPUs. `libHetMP`*'s goal is to automatically determine where to place parallel computation in heterogeneous CPU systems to maximize performance.* `libHetMP` incorporates two new components into the OpenMP runtime. First, it provides the mechanisms necessary to execute OpenMP-parallelized computation across heterogeneous-ISA CPUs, including migrating threads to different nodes and distributing parallel work from work sharing regions to those threads. Second, `libHetMP` automates work distribution decisions by measuring system performance metrics. In particular, `libHetMP` analyzes DSM activity over the interconnect between CPUs and performance counters. `libHetMP` implements a new loop iteration scheduler, called the *HetProbe* scheduler, that uses those metrics to make work distribution decisions. Using these metrics, the HetProbe scheduler either utilizes cross-node execution or selects a single node on which to execute. `libHetMP` alleviates developers from having to manually configure applications for each new hardware setup, e.g., new CPUs or different types of interconnects.

3.1 OpenMP Across Heterogeneous-ISA CPUs

In order to support existing OpenMP applications, `libHetMP` must be compatible with existing semantics and therefore abstract all mechanisms behind runtime entry points. `libHetMP` inserts migration points when starting thread teams in order to execute OpenMP `parallel` regions across nodes. After thread migration, `libHetMP` internally separates runtime chores (work distribution, synchronization, performance monitoring) into per-node and global operations in order to minimize DSM traffic generated by the runtime itself. The runtime must 1) organize and place threads across nodes to minimize cross-node communication and 2) distribute work, including measuring application performance across and within nodes to adjust distribution decisions.

Cross-Node Execution. When starting a `parallel` region, `libHetMP` organizes threads into a *thread hierarchy* to break synchronization down into per-node and global operations, significantly reducing the amount of cross-node traffic caused by synchronization. At application startup, `libHetMP` queries the system to determine each node's characteristics,

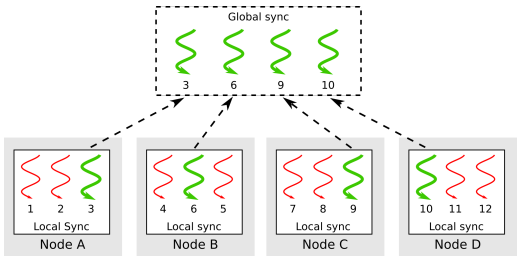


Figure 3. libHetMP's thread hierarchy. In this setup, libHetMP has placed 3 threads (numbered 1-12) on each node. For synchronization, threads on a node elect a leader (green) to represent the node at the global level. Non-leader threads (red) wait for the leader using local synchronization to avoid cross-node data accesses.

i.e., type and number of CPUs available, and uses this information to initialize the thread hierarchy. As threads are spawned or re-initialized, libHetMP calls the OS's thread migration function to physically place threads according to the hierarchy. The runtime migrates threads to nodes based on how many threads are executing the parallel region and how many CPUs are available on each node – for example, in a setup containing a 16-core Xeon and a 96-core ThunderX, libHetMP spawns and places 16 and 96 threads, respectively, for a total of 112 threads. Internally, libHetMP initializes per-node data structures like loop iteration scheduler metadata, barriers and counters based on the hierarchy. libHetMP also initializes global variants of these data structures for cross-node synchronization (see below). libHetMP allows re-configuring the thread hierarchy between parallel regions, which is useful for dynamically adjusting parallel execution.

Synchronization. libHetMP uses the thread hierarchy for many types of synchronization, including barriers, reductions and work distribution. libHetMP builds upon previous work by Lyerly et al. [37], which refactors OpenMP execution for DSM on homogeneous clusters. Figure 3 illustrates using the thread hierarchy for synchronization. When synchronizing, threads on each node elect a *node leader* to act on their behalf at the global level and the remaining non-leader threads wait at per-node barriers. The node leaders act as representatives for the node and communicate through global data structures using DSM. The leader/non-leader designation significantly reduces cross-node communication as most threads do not touch global data (in the Xeon/Cavium setup, only 2 threads touch global data instead 112). For example, when executing parallel reductions the first thread on a node to arrive is elected leader and waits for all non-leader threads to make their local data available for reduction. Once the leader has reduced data from all threads on its node, it produces the node's data for the final reduction at the global level. A global leader, elected in the same fashion, is responsible for reducing data from all nodes.

Workload Distribution. OpenMP defines several *loop iteration schedulers* that affect how iterations of a work-shared parallel loop are mapped to threads. The default loop iteration schedulers (static, dynamic) implement several strategies with the goal of evenly partitioning work to avoid overloaded straggler threads from harming performance. libHetMP provides the ability to distribute iterations across nodes by extending these schedulers to account for heterogeneity and to efficiently synchronize how threads grab iterations. Due to limitations in each (see below), libHetMP introduces the HetProbe scheduler for automatic iteration distribution in consideration of CPU and interconnect.

libHetMP assumes each node in the system contains a set of homogeneous CPU cores with identical micro-architecture and cache coherence. To quantify performance differences between nodes, libHetMP defines a *core speed ratio* (CSR) to rank the relative compute capabilities of individual CPU cores on one node versus another. For example, a Xeon core with a core speed ratio of 3:1 compared to a ThunderX core means the Xeon core is considered 3x faster than a ThunderX core and threads running on the Xeon will get 3x as many loop iterations as threads on the ThunderX. Note that CSRs are assigned to each work sharing region, as applications may have multiple work sharing regions that exhibit different performance characteristics.

Cross-node static scheduler. OpenMP's static scheduler evenly partitions loop iterations among threads, assigning each thread the same number of iterations. The scheduler implicitly assumes all CPUs are equal and all loop iterations perform the same amount of work. Rather than considering all threads equal, libHetMP allows developers to specify per-node CSRs to skew work distribution for threads on different nodes. The challenge, however, is that developers must manually discover the ideal CSR for each work sharing region and hardware configuration through extensive profiling.

Cross-node dynamic scheduler. With OpenMP's dynamic scheduler, threads continuously grab user-defined batches of iterations from a global work pool using atomic operations on a global counter. This scheduler targets work sharing regions where individual loop iterations perform varying amounts of work. libHetMP optimizes grabbing batches using the thread hierarchy – threads first attempt to grab iterations from a node-local work pool instantiated during team setup. If the local pool is empty, the thread grabbing iterations is elected leader and transfers iterations from the global pool to the per-node pool. Because the leader represents the entire node, it grabs a batch of iterations for each thread executing on the node. This reduces the number of threads accessing the global pool and thus the amount of global synchronization required for work distribution.

While not traditionally meant for load balancing on heterogeneous systems, the dynamic scheduler can load balance work distribution based on the compute capacity of CPUs in the system. However, continuous synchronization both at

the local and global level to grab batches of work can negatively impact performance, especially with small batch sizes. Users must again profile to determine the ideal per-region and per-hardware batch size. Non-deterministic mapping of loop iterations to threads can also cause "churn" in the DSM layer for applications that execute the same work sharing region multiple times. With deterministic mapping of iterations to threads, data may settle after the first invocation as nodes acquire the appropriate pages and permissions. The `dynamic` scheduler prevents data from settling on nodes.

The main problem with the default schedulers is that users must extensively profile to find the best workload distribution configuration in a large state space, i.e., determine CSRs or batch sizes for each individual work sharing region on every new heterogeneous platform. Additionally, if cross-node execution is not beneficial for a work sharing region due to large DSM overheads, users must profile to determine the best CPU for single-node execution and manually reconfigure the thread team (including the thread hierarchy) to only execute work-sharing regions on the selected CPU.

HetProbe scheduler. To avoid the tuning complexity of the default schedulers, `libHetMP` introduces a new *heterogeneous probing* or HetProbe scheduler, for automatically configuring execution of parallel computation. Developers only need to specify the HetProbe scheduler like other OpenMP schedulers, e.g., adding a `schedule(hetprobe)` clause to a work sharing region, and `libHetMP` will transparently handle distributing loop iterations to available CPUs. Developers do not need to reason about DSM overheads or performance characteristics of individual nodes. The HetProbe scheduler executes a small number of iterations across both nodes, called the *probing period*, during which it measures per-core execution time, cross-node page faults and performance counters to analyze a work sharing region's behavior. The HetProbe scheduler is designed to optimize the performance of work sharing regions with regular loops, where the behavior of one loop iteration is a good predictor for the behavior of other iterations. The HetProbe scheduler uses the performance analysis information gathered during the probing period to distribute the remaining iterations as described in Section 3.2.

The HetProbe scheduler must be precise when distributing iterations for the probing period in order to accurately evaluate system performance. First, the scheduler issues a constant number of loop iterations to each thread, regardless of node, in order to compare the execution time of equal amounts of work on each CPU. Second, the scheduler must deterministically issue iterations, so that threads executing a work sharing region multiple times receive the same batch of iterations across invocations to account for the aforementioned data settling effect. If the HetProbe scheduler non-deterministically distributes probe iterations, data might unintentionally churn and cause falsely higher DSM overheads.

`libHetMP` also implements a *probe cache* for applications that execute a work sharing region multiple times. This has two benefits – first, it allows the runtime to reuse previously calculated statistics and workload distribution decisions from previous probing periods to avoid probing overheads. Second, `libHetMP` uses multiple probing results to smooth out measurement variation for shorter-running work sharing regions. `libHetMP` uses an exponential weighted moving average for measurement statistics, which favors more recent measurements and quickly converges on accurate values. `libHetMP` uses this type of average because initial probing values for regions may be inaccurate due to the DSM layer initially replicating data across nodes whereas subsequent executions may incur fewer DSM costs.

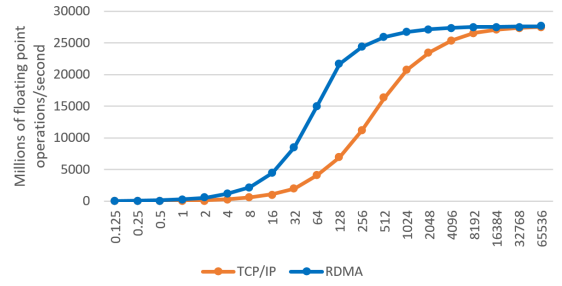
3.2 Workload Distribution Decisions

The HetProbe scheduler uses the execution time, page faults and performance counters measured during the probing period to determine where to execute parallel work. Specifically, the HetProbe scheduler answers three questions:

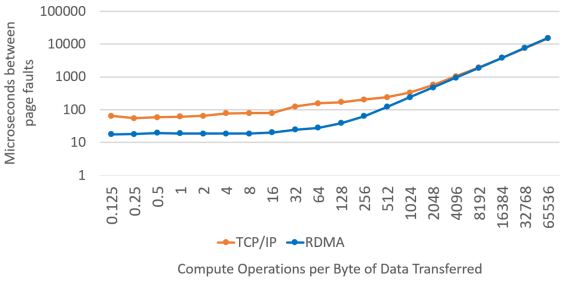
1. Should the runtime leverage multiple nodes for parallel execution? While coupling together multiple CPUs provides more theoretical computational power, not all applications benefit from cross-node execution. As mentioned in Section 2 there is a significant cost for on-demand data marshaling and page coherency across nodes. To understand DSM overheads, we ran a microbenchmark that varies the number of compute operations executed per byte of data transferred over the interconnect. Because there are no server-grade heterogeneous-ISA CPUs integrated by point-to-point interconnects, we approximate a system using the experimental setup shown in Table 1 and evaluated the DSM layer using two protocols, TCP/IP and RDMA.

The microbenchmark spawns one thread for every core in every node in the system. It then runs a control loop that stresses each node (i.e., each $(architecture, interconnect)$ pair) connected to the *source* node, i.e., the Xeon, because it runs the single-threaded portion of applications. At the start of the control loop, the source node threads initialize memory by touching all data pages to force the DSM protocol to bring all pages back to the source node's memory. The control loop then releases the other node's threads (ThunderX) to begin timed execution. Each ThunderX thread touches non-overlapping sets of pages to force the DSM protocol to transfer them to ThunderX memory. Finally, the ThunderX threads perform varying amounts of compute operations per page transferred. The microbenchmark calculates operations/second (incorporating the DSM costs) by timing how long it takes to execute the loop to determine the break-even point where cross-node execution is beneficial.

Figure 4a shows the compute throughput in millions of floating point operations per second when varying the number of compute operations per byte of data transferred over the interconnect. Figure 4b shows the average page fault



(a) Floating point operations per second



(b) Page fault period, i.e., microseconds between faults

Figure 4. Performance metrics observed when varying the number of compute operations per byte of data transferred over the interconnect. For example, a 16 on the x-axis means 16 math operations were executed per transferred byte or 65536 operations per page.

Table 1. Experimental setup

Description	Xeon 2620v4	ThunderX
Vendor	Intel	Cavium
Cores	8 (16 HT)	96 (2 x 48)
Clock (GHz)	2.1 (3.0 boost)	2.0
LLC Cache	L3 - 16MB	L2 - 32MB
RAM (Channels)	32 GB (2)	128 GB (4)
Interconnect	Mellanox ConnectX-4 56Gbps	

period, i.e., elapsed time between subsequent page faults. Intuitively, as threads perform more computation per byte transferred, the computation is able to amortize the DSM costs and reach peak throughput. As shown in Figure 4b, there are significant latency differences between RDMA and TCP/IP. Page faults using RDMA cost around 30 microseconds, whereas they cost 90 and 120 microseconds for the Xeon and Cavium servers, respectively, with TCP/IP. Thus, the amount of computation needed to amortize DSM costs when using TCP/IP is significantly higher than RDMA.

To determine if cross-node execution is beneficial, the HetProbe scheduler calculates the page fault period by measuring execution times and number of faults. The break-even point when cross-node execution becomes beneficial can be seen in Figure 4a when the microbenchmark is close to maximum throughput: above 512 operations/byte for RDMA, 32768 operations/byte for TCP/IP. Correlating these values to Figure 4b, the runtime uses a threshold of 100 μs/fault for RDMA and 7600 μs/fault for TCP/IP to determine whether there is enough computation to amortize DSM costs and benefit from executing across multiple CPUs. As faulting latency drops (e.g., if CPUs share physical memory), fewer compute operations are needed to amortize cross-node memory access latencies. When the interconnect between CPUs changes, this microbenchmark can be re-used as a tool to automatically determine the threshold value of when cross-node execution becomes beneficial.

2. If utilizing cross-node execution, how much work should be distributed to each node? As mentioned previously, during the probe period the runtime measures the execution time of a constant number of iterations on each core in the system. The HetProbe scheduler uses this information to directly calculate the core speed ratios of each node and skew the distribution of the remaining loop iterations.

3. If not utilizing cross-node execution, on which node should the work be run? Determining on which node an application executes best involves understanding how the application stresses the architectural properties of each CPU. Performance counters provide insights into how applications execute and what parts of the architecture bottleneck performance. For our setup, the ThunderX has a much higher degree of parallelism versus the Xeon, meaning it has a much higher theoretical throughput for parallel computation. However, the biggest challenge in utilizing all 96 cores is being able to supply data from the memory hierarchy. Although the ThunderX uses quad-channel RAM (with twice the bandwidth of the Xeon), it only has a simple two level cache hierarchy versus the Xeon's much more advanced (and larger per-core) three level hierarchy. If an application exhibits many cache misses, it is unlikely to fully utilize the 96 available cores and would be better run on the Xeon. The HetProbe scheduler measures cache misses per thousand instructions during the probing period to determine how much the work-sharing region stresses the cache hierarchy (users can specify any performance counters prudent for their hardware). We experimentally determined a threshold value of three cache misses per thousand instructions – below the threshold and the application can take advantage of the ThunderX's parallelism, but above the threshold the ThunderX's CPUs will continuously stall waiting on the cache hierarchy. Note that the HetProbe scheduler must use performance counters and cannot simply use execution times from the probing period to decide on a node; the probing period measures execution times with DSM overheads that are not present when executing only on a single node.

Once a node has been chosen, the HetProbe scheduler falls back to existing OpenMP schedulers for single-node work distribution. Currently it defaults to the static scheduler, but this is configurable by the user. Additionally, `libHetMP` joins threads on the unused node to avoid unnecessary cross-node synchronization overheads. For example, if not using the

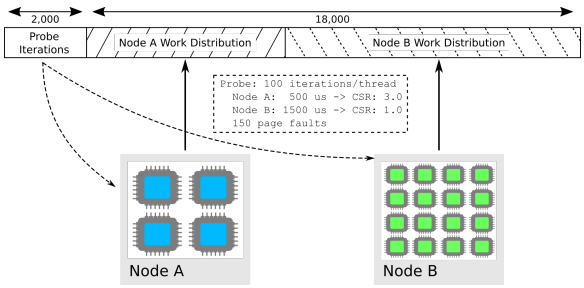


Figure 5. HetProbe scheduler. A small number of probe iterations are distributed at the beginning of the work-sharing region to determine core speed ratios of nodes in the system. Using the results, the runtime decides either to run all iterations on one of the nodes or distribute work across nodes according to the calculated core speed ratio (shown here).

ThunderX there is no reason to keep 96 threads alive simply to join at end-of-region barriers.

Figure 5 shows an example of a work sharing region with 20000 loop iterations executing using the HetProbe scheduler. The first 2000 iterations are used for the probing period and each of the 20 cores across both nodes is given an equal share of 100 iterations. Importantly, the probing period is performing useful work, albeit in a potentially unbalanced way. After the probing period, `libHetMP` measures that Node A's cores executed 100 iterations in 500μs whereas Node B's cores executed 100 iterations in 1500μs. The HetProbe scheduler determines that Node A's cores are 3x faster than Node B's cores for this work sharing region, meaning threads on Node A should get 3x more iterations than threads on Node B to evenly distribute work (the CSR is set to 3:1 for Nodes A and B, respectively). In this example, the HetProbe scheduler determined that cross-node execution was beneficial (see Section 3.2). For the remaining 18000 iterations, each thread on Node A receives 1929 iterations and each thread on Node B receives 643. Thus the HetProbe scheduler automatically determines the relative performance of heterogeneous CPUs through online profiling and distributes the remaining work accordingly. Note that if cross-node communication was deemed too costly, the remaining 18000 iterations would all be distributed to either Node A or Node B.

4 Implementation

`libHetMP` is built on top of GNU `libgomp`, the OpenMP runtime used by `gcc`. It adds 5,145 lines of code, primarily to implement the thread hierarchy (and all associated machinery), runtime measurement and dynamic work distribution. Because Popcorn Linux's compiler is built on `clang` which emits API calls the `libiomp` runtime, `libHetMP` includes a small shim layer to forward `libiomp` function calls to `libgomp`. However, none of the ideas presented are specific to either OpenMP implementation. Because page faults are transparent to the application, `libHetMP` reads page fault counters from a `proc` file exposed by Popcorn Linux. Currently Popcorn Linux does not support runtime performance counter collection; to work around this, we collected per work sharing region performance counter data offline and fed it to `libHetMP` via environment variables to make node selection decisions. For applications with multiple work-sharing regions, the user currently manually specifies which region should be probed to decide whether cross-node execution is beneficial. This is an optimization to avoid unnecessary probing, as our current benchmarks did not have different individual regions that benefited from different work distribution decisions. We instrumented `libHetMP` to record the time spent in each work sharing region and selected the longest running region as the probing region. The user instructed `libHetMP` to use this region as the probing region by passing a compiler-constructed region identifier (constructed from the filename, containing function and line number of the work sharing directive) via environment variables. This could be automated by `libHetMP` by running the application for a small period of time and querying the probe cache to select the longest running region. We leave these engineering tasks as future work.

5 Evaluation

When evaluating `libHetMP` we asked whether 1) is `libHetMP` able to efficiently leverage the compute capabilities of asymmetric server-grade heterogeneous CPUs? 2) is `libHetMP`'s HetProbe scheduler able to accurately measure runtime behavior and make sound workload distribution decisions? Specifically, can the HetProbe scheduler accurately determine if cross-node execution is beneficial, distribute appropriate amounts of work to each node, and select the best CPU for single-node execution? and 3) which schedulers are best suited for which types of runtime behaviors?

Experimental Setup. We evaluated `libHetMP` using the experimental setup in Table 1, which approximates our envisioned tightly-coupled platform. Because no existing systems integrate heterogeneous-ISA CPUs via point-to-point connections, we approximate one by connecting two servers with high-speed networking. Our setup includes an Intel Xeon server with a modest number of high-powered cores and a Cavium ThunderX server with a large number of lower-performance cores. The machines are connected via 56Gbps InfiniBand, which provide low latency and high throughput. We use the RDMA protocol for all experiments except where mentioned due to its significantly lower latency. Both machines run the latest version of Popcorn Linux; the Xeon server uses Debian 8.9 while the ThunderX server uses Ubuntu 16.04. Popcorn Linux's compiler is built on `clang`/LLVM 3.7.1, and `libHetMP` is built on `libgomp` 7.2.0.

Benchmarks. We selected 10 benchmarks from three popular benchmarking suites – The Seoul National University [50] C/OpenMP versions of the NAS Parallel Benchmarks [6], PARSEC [10] and Rodinia [14]. These benchmarks

represent HPC and data mining use cases and exhibit a wide variety of computational patterns on which to evaluate `libHetMP`. All benchmark results are the average of 3 runs (execution times were stable across runs). All benchmarks were compiled with `-O2` except CG and cfd, which crashed Popcorn's compiler unless compiled with `-O0`. We also used `-fopenmp-use-tls` to enable Linux-native TLS.

Work Distribution Configurations. We evaluated running benchmarks using several workload configurations. *Xeon* represents running the benchmark entirely on the Xeon – serial phases run on a single Xeon core and work-sharing regions use the Xeon's 16 threads. *ThunderX* is similar – serial phases run on a single ThunderX core and work-sharing regions use the ThunderX's 96 cores. *Ideal CSR* executes across both CPUs – serial phases run on a Xeon core and work-sharing regions always split loop iterations across the Xeon and ThunderX (112 total threads) using the static scheduler. The scheduler skews distribution using the CSRs in Table 2. The CSRs were gathered from runs with the HetProbe scheduler and manually supplied via environment variables. *Cross-Node Dynamic* is identical except it uses the hierarchy-based dynamic scheduler described in Section 3.1. We experimentally determined the best chunk size for each benchmark; most benchmarks performed better with smaller sizes, i.e., finer-grained load balancing. *HetProbe* is again identical except it uses the HetProbe scheduler. HetProbe uses both CPUs during the probing period and then decides whether cross-node execution is beneficial. If so, it uses measured execution time to calculate CSRs (Table 2) to skew loop iteration distribution for the remaining iterations. If not, it selects the best CPU and falls back to OpenMP's original static scheduler on a single node; threads on the not-selected node are joined to avoid unnecessary synchronization. The probe period was configured to use 10% of available loop iterations. For benchmarks where cross-node execution was beneficial, probing overhead was determined by comparing the difference in performance between Ideal CSR and HetProbe. For benchmarks where it was not beneficial, probing overhead was determined by comparing the delta between the best single-node performance (either Xeon or ThunderX) and HetProbe. The HetProbe scheduler probed for up to 10 invocations of a given work-sharing region (using an exponential weighted moving average to smooth out measurements), after which it re-used existing measurements from the probe cache. For several benchmarks, the HetProbe scheduler chose single-node execution on the ThunderX. As a comparison point, "HetProbe (force Xeon)" shows the same results except forcing the HetProbe scheduler to use single-node execution on the Xeon; these results are explained below.

Results. Table 3 shows the total benchmark execution times, including both serial and parallel phases, on the Xeon. Figure 6 shows the speedup normalized to homogeneous Xeon execution for each of the aforementioned configurations. The benchmarks can broadly be classified into two categories: those that benefit from cross-node execution and those that do not. blackscholes, EP-C, kmeans and lavaMD fall into the former category whereas the others fall into the latter. Across benchmarks that benefit from multi-node execution, all but blackscholes achieve the highest speedup under Cross-Node Dynamic. This is because with a granular chunk size, work is distributed across nodes in an almost perfect balance. Additionally, due to the thread hierarchy there is significantly reduced global synchronization and threads grab work from a local work pool the majority of the time. Across these four benchmarks, Cross-Node Dynamic yields a geometric mean speedup of 2.68x. Ideal CSR is 12.5% faster for blackscholes and close behind Cross-Node Dynamic for the other three, achieving a geometric mean speedup of 2.55x. Finally, HetProbe is slightly slower than the other two cross-node configurations, achieving a geometric mean speedup of 2.4x. This is because the probe period runs a constant number of iterations for all cores leading to an initial workload imbalance. Additionally, measurement machinery (timestamps, parsing the `proc` file for DSM counters) and probe cache synchronization add extra overheads. For these four benchmarks, probing overhead is equal to the difference between Ideal CSR and HetProbe, as they are functionally equivalent after probing. HetProbe adds 5.2%, 5.3%, 11.5% and 2.8% overhead for blackscholes, EP-C, kmeans and lavaMD, respectively, for a geometric mean overhead of 5.5%. This demonstrates the HetProbe scheduler provides competitive performance with minimal overheads for benchmarks that benefit from cross-node execution.

For benchmarks that do no scale across nodes, however, the Ideal CSR and Cross-Node Dynamic configurations significantly degrade performance with geometric mean slowdowns of 3.63x and 5.89x, respectively. This is due to DSM – threads spend significant time waiting for pages from other nodes, which also forces application threads on other nodes to be time-multiplexed with DSM workers. There is not enough computation to amortize DSM page fault costs over

Table 2. Core speed ratios calculated by HetProbe scheduler. Used by Ideal CSR and HetProbe configurations. Without the HetProbe scheduler, developers would have to manually determine these values via extensive profiling.

Benchmark	Core speed ratio – Xeon : ThunderX
blackscholes	3 : 1
EP-C	2.5 : 1
kmeans	1 : 1
lavaMD	3.666 : 1

Table 3. Baseline execution times in seconds when run on Xeon with 16 threads using the static scheduler

Benchmark	Time	Benchmark	Time
blackscholes	85.76	kmeans	989.77
BT-C	310.08	lavaMD	104.52
cfd	76.47	lud	258.75
CG-C	71.36	SP-C	210.57
EP-C	32.00	streamcluster	67.86

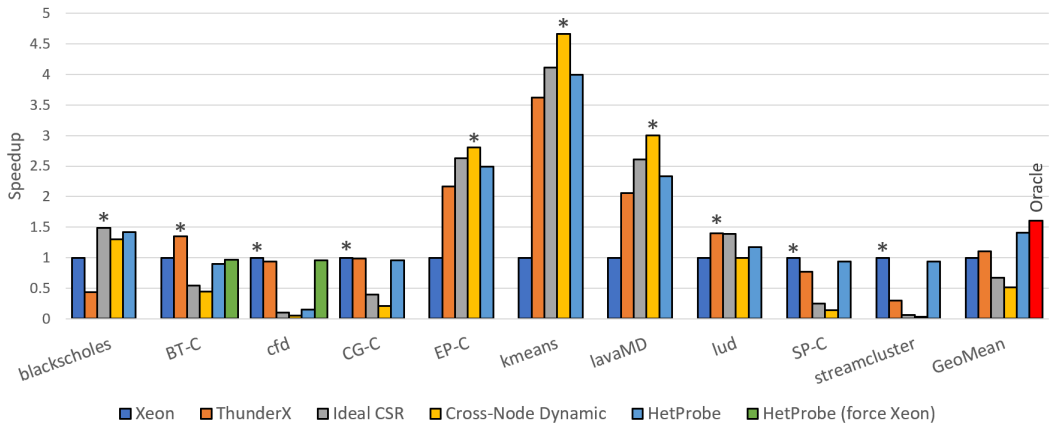


Figure 6. Speedup of benchmarks versus running homogeneously on Xeon (values less than one indicate slowdowns). Asterisks mark the best workload distribution configuration for each benchmark. "Cross-Node Dynamic" provides the best performance across applications that benefit from leveraging both CPUs (blackscholes, EP-C, kmeans, lavaMD), but causes significant slowdowns for those that do not. "HetProbe" achieves similar performance to Ideal CSR and Cross-Node Dynamic for these four applications but falls back to a single CPU for applications with significant DSM communication and hence worse cross-node performance. For geometric mean, "Oracle" is the average of the configurations marked by asterisks, i.e., what a developer who had explored all such possible workload distribution configurations through extensive profiling would choose.

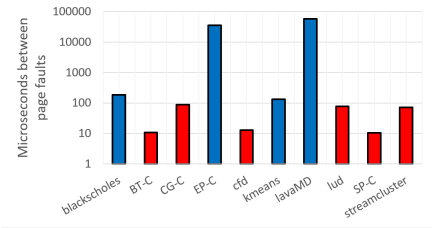


Figure 7. Page fault periods determining whether cross-node execution is beneficial. Red bars (cross-node not profitable) are below the RDMA threshold indicated in Section 3, blue are above.

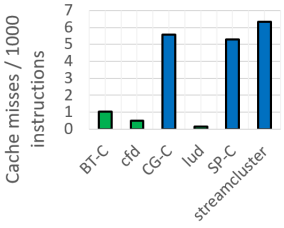


Figure 8. Cache misses for applications not executed across nodes. Green bars (including lud) indicate the application was run on the ThunderX, blue were run on the Xeon.

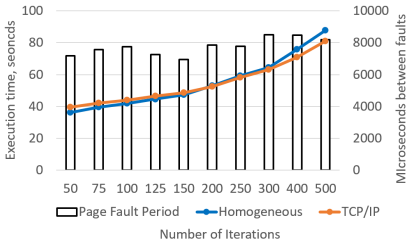


Figure 9. Execution time (lines, left axis) and page fault period (bars, right axis) for blackscholes. "Homogeneous" refers to Xeon configuration, "TCP/IP" refers to using HetProbe over TCP/IP.

the network. The Cross-Node Dynamic scheduler is exclusively worse than the Ideal CSR scheduler due to additional work distribution synchronization caused by threads repeatedly grabbing batches of iterations. The HetProbe scheduler, however, successfully avoids cross-node execution for these benchmarks by measuring the page fault period and determining cross-node execution to not be beneficial (geometric mean slowdown of 39%, or 2.4% without cfd). Figure 7 shows measured page fault periods for each application; applications with a period below 100μs were considered not profitable for cross-node execution.

For applications deemed not beneficial to execute across nodes due to high DSM overheads, the HetProbe scheduler utilized cache misses per 1000 instructions to determine whether to execute work-sharing regions on the Xeon or ThunderX. As shown in Figure 8, there is a clear separation between applications that benefit from the ThunderX's high parallelism (BT-C, cfd, lud) and those that are bottlenecked by memory accesses (CG-C, SP-C, streamcluster). When selecting a node, the HetProbe scheduler used a threshold value of three misses per thousand instructions, placing BT-C, cfd and lud on the ThunderX and the others on Xeon (cfd has special behavior, see below). For the three benchmarks placed on Xeon, probing overhead is equivalent to the difference between Xeon and HetProbe since HetProbe degrades to Xeon after probing. The probing period adds 4.8%, 6.6% and 7.1% for CG-C, SP-C and streamcluster, respectively, for a geometric mean overhead of 6.1%. This shows performance close to single-node execution on the Xeon, meaning the probing period has minimal impact on performance.

Looking closer at cfd and CG-C, these applications have roughly the same performance on Xeon and ThunderX but are vastly different in cache miss behavior. Even more interestingly, the HetProbe scheduler places cfd on the ThunderX although the optimal choice would be on the Xeon. This is due to the fact that although cfd's parallel region runs faster on the ThunderX (74.58 seconds on Xeon, 66.79 seconds on

ThunderX), it has a long serial file I/O phase that runs significantly faster on Xeon (1.83 seconds on Xeon, 13.72 seconds on the ThunderX), leading the benchmark's overall execution time to be faster on the Xeon. This file I/O phase also explains the disparity in cache misses between benchmarks – cfd's parallel region has a low number of cache misses, but the benchmark's execution time is heavily impacted by file operations whereas CG-C does not perform file I/O and has a large number of cache misses.

Interestingly for BT-C, cfd and lud, executing parallel regions on the ThunderX achieved worse than expected performance due OS limitations. Popcorn Linux's kernel currently only supports spawning threads on the node on which the application started, meaning one thread must remain on the Xeon even when work-sharing regions execute on the ThunderX. Each of these benchmarks executes hundreds to thousands of work-sharing regions (and their associated implicit barriers), causing significant cross-node synchronization. As a comparison point for BT-C and cfd, we ran an additional experiment to force the HetProbe scheduler to select the Xeon for single-node execution; it added 3.2% and 4.2% probing overhead, respectively. lud is an interesting case – the HetProbe scheduler decides cross-node execution is not profitable and runs work sharing regions on the ThunderX. The aforementioned OS limitation impacts HetProbe's performance enough that Ideal CSR actually achieves 20% better performance than HetProbe (although still worse than running solely on the ThunderX). We expect that when Popcorn Linux allows spawning threads on remote nodes, `libHetMP` will be able to more efficiently leverage both machines.

It is important to note that none of Xeon, ThunderX, Ideal CSR or Cross-Node Dynamic perform best in all situations, clearly illustrating the need for HetProbe. As shown in Figure 6, HetProbe provides the best performance out of all evaluated configurations across all benchmarks with a geometric mean performance improvement of 41% (ThunderX provides an 11% improvement). In contrast, Ideal CSR causes a slowdown of 49% and Cross-Node Dynamic causes a 96% slowdown, highlighting the importance of communication traffic when distributing computation. As a comparison point, "Oracle" shows that developers could obtain a geometric mean speedup of 60% if they had extensively profiled all configurations and selected the best for all benchmarks. As Popcorn Linux matures, HetProbe will be able to more closely match the Oracle, as the aforementioned limitation has a significant impact on HetProbe's performance.

What types of applications benefit from cross-node execution? The four applications that benefit from cross-node execution have a high enough compute to cross-node communication ratio to leverage the compute resources of multiple CPUs. blackscholes has an initial data transfer period but repeats computation on the same data, allowing it to settle on nodes (blackscholes also has a lengthy file I/O phase that benefits from the Xeon's strong single-threaded performance). EP-C performs completely local computation (including heavy use of thread-local storage) with a single final reduction stage. lavaMD computes particle potentials through interactions of neighbors within a radius, meaning multiple threads re-use the same data brought across the interconnect. Similarly, kmeans alternatively updates cluster centers and cluster members – all threads on a node alternate between scanning the cluster member and cluster center arrays, re-using pages brought over the interconnect.

Benchmarks that do not benefit cannot amortize data transfer costs. For example, BT-C and SP-C access multi-dimensional arrays along different dimensions in consecutive work sharing regions, causing the DSM to shuffle large amounts of data between nodes. Other benchmarks have little data locality – CG-C and streamcluster calculate a set of results and then access them in irregular patterns using an indirection array. This behavior causes extensive latencies for local cache hierarchies, let alone DSM. lud's work-sharing region sequentially accesses an array, but does not perform enough computation per byte to amortize DSM costs. Additionally, there is a large amount of "false sharing" where threads on different nodes write to independent parts of the same page. False sharing can be avoided by the use of a multiple-writer protocol such as lazy-release consistency [4].

An interesting observation that arises from measuring the page fault period is that while the metric provides a sound threshold for determining whether cross-node execution will be beneficial, it is not a good indicator of overall performance gains. For example, while kmeans' page fault period is slightly over the threshold (130 μs period), it benefits the most from cross-node execution out of all benchmarks. This is because it has a high level of inter-thread *data reuse*. As mentioned previously, all threads scan the same array in a superstep of the algorithm, meaning all threads reuse data brought over from a page fault (in addition to being extremely efficient on the cache-starved ThunderX). This is in contrast to lavaMD where only a subset of threads working on adjacent regions reuse migrated pages. `libHetMP` only observes DSM traffic for the entire application and does not measure the level of data reuse for migrated pages. This leads us to believe that the current thresholding mechanism will degrade when threads executing a work sharing region have skewed page fault behavior, e.g., a few threads cause the majority of page faults, biasing the work sharing region's average page fault period. In such a case `libHetMP` may determine there is too much communication for cross-node execution, even though it may still be beneficial. However, because we focus on applications with regular work sharing regions, we did not observe this skewed page fault behavior. This also highlights the importance of directly measuring the relative performance of the CPUs to make workload distribution decisions for cross-node execution.

What applications benefit from Ideal CSR versus Cross-Node Dynamic? Three of the four benchmarks that

benefit from cross-node execution achieve the best performance with Cross-Node Dynamic due to fine-grained load balancing. For blackscholes, however, Ideal CSR achieves better performance. This is due to pages settling into a steady state after an initial page shuffle. Threads receiving the same loop iterations across multiple invocations of the work sharing region access the same data, thus all data pages required by threads are already mapped to the appropriate node. With Cross-Node Dynamic, however, threads receive different loop iterations across separate executions, meaning pages containing results must be continually shuffled across nodes. This settling behavior is why the HetProbe scheduler deterministically distributes iterations for the probing period.

Case Study: TCP/IP. In order to evaluate the effectiveness of the HetProbe scheduler for different types of interconnects, we ran blackscholes with varying number of iterations (more iterations means more compute operations per byte since blackscholes' data settles after the first iteration) using the TCP/IP protocol described in Section 3.2. Figure 9 shows the execution time when running homogeneously on the Xeon versus cross-node execution (lines) and the page fault period of each cross-node run (bars). We use a page fault period of 7600μs to determine whether cross-node execution will be beneficial when using TCP/IP. The results are somewhat noisy (TCP/IP tends to have more variable latencies) but consistent with expectations – only after the page fault period climbs above 8000μs does cross-node execution pay off. Thus we conclude using page fault periods as the determining factor for cross-node execution is applicable for different types of interconnects.

Limitations. There are number of ways in which `libHetMP` can be extended. First, the HetProbe scheduler is designed for work sharing regions with loops where each loop iteration performs a constant and equal amount of work. This assumption allows the HetProbe scheduler to make workload distribution decisions by monitoring the behavior of a small number of probe iterations. For irregular applications such as graph traversal algorithms [32], however, predicting cross-node DSM traffic becomes significantly more difficult and potentially requires co-designing DSM and parallel programming runtimes to minimize communication. One approach to dealing with these types of regions would be to logically break a work sharing region into multiple smaller work-sharing regions, each with their own probing and workload distribution decisions. Another possibility would be to continuously monitor page faults during the work sharing region and fall back to single node execution if the number of page faults begins to rise. Conversely, if the number of page faults begins to drop, the HetProbe scheduler could dynamically bring extra threads on another node online. In general, `libHetMP` and the HetProbe scheduler could be extended to provide a more dynamic distribution of parallel work.

`libHetMP` also currently focuses on achieving maximum performance but not energy efficiency. The first-generation ThunderX CPUs consume large amounts of power, meaning that even though cross-node execution may provide the best performance oftentimes the heterogeneous setup consumes more energy than running solely on one node. Optimizing OpenMP execution for different efficiency metrics may yield different workload distributions, especially as the system architecture (CPUs, interconnect) changes.

`libHetMP` could be extended to handle systems with three or more nodes. The microbenchmark described in Section 3.2 can be used to determine when cross-node execution becomes profitable for each $(architecture, interconnect)$ pair attached to the system (i.e., each *node*). This break-even point is different for every node and decisions about which nodes to use can be made independently from one another. For example, consider a system with nodes A and B with break-even points of 100 us/fault and 200 us/fault, respectively. If `libHetMP` measured a page fault period of 150us/fault for a given work-sharing region, then the HetProbe scheduler could choose to distribute work to node A but not use node B. In this way, the HetProbe scheduler can choose whether to utilize each individual node and distribute iterations to the enabled nodes according to their relative performance in the probing region.

6 Related Work

Parallel Programming Models and Frameworks. Shared-memory parallel programming models like OpenMP [42] and Cilk [11] provide source code annotations to automate parallel computation, but do not support execution across cache-incoherent, heterogeneous-ISA CPUs. MPI [23] gives developers low-level primitives to distribute execution, manage separate physical memories and marshal memory between heterogeneous-ISA CPUs. However for asymmetric CPUs, developers must manually assign parallel work and transfer required data to maximize performance, leading to complex and verbose applications with static, non-portable workload distribution decisions. Cluster OpenMP [25] is a now-defunct commercial product attempting to replace hierarchical MPI + OpenMP parallelism by providing shared memory semantics using DSM on networked homogeneous machines. PGAS frameworks like UPC [15], X10 [13] and Grappa [39] support cross-node execution and memory accesses, but do not support sharing data across ISAs and changing workload distribution decisions in light of system characteristics is cumbersome (data is not migrated between nodes for locality). Charm++ [27] is an object-oriented approach to sharing data between (potentially distributed) processes, but does not support load balancing across heterogeneous-ISA CPUs. Cluster frameworks like SnuCL-D [29] and OmpSs [12] provide coarse-grained work distribution by assigning multiple

independent parallel computations to individual heterogeneous processors. They do not consider fine-grained worksharing of a single parallel computation and require developers to specify data movement. libHetMP automatically distributes work in consideration of platform characteristics and leverages transparent and on-demand DSM to manage memory consistency for flexibility and programmability.

CPU/GPU Work Partitioning. Several works explore work distribution in CPU/GPU systems. Qilin [36] is a compiler and runtime that enables CPU/GPU workload partitioning but requires developers to rewrite computation using a new API. Unlike libHetMP, Qilin does not make distribution decisions online but must profile multiple full executions before determining the optimal workload split. Kofler et al. [31] present a machine learning approach to determining workload distribution, but require sophisticated analyses with a custom compiler and the machine learning model must be retrained for each new hardware configuration. Similarly, Grewe and O'Boyle [22] present a machine learning approach that requires per-system retraining. Scogland et al. [49] present CPU/GPU workload distribution approaches for accelerated OpenMP. However their approach only works for dense array-based computations and developers must manually specify data movement between devices. All of these approaches are limited by the visible split in CPU and GPU memory and require developer to marshal data. Additionally, none of these approaches provide optimized cross-node synchronization primitives and none consider situations where cross-node execution may not be beneficial.

Single-ISA Scheduling. There are a number of schedulers designed to improve task-parallel workloads (as opposed to data-parallel workloads targeted by libHetMP) on single-ISA heterogeneous systems, e.g., ARM big.LITTLE [21]. The Lucky scheduler [43] measures the energy efficiency of multiprogrammed workloads via performance counters and uses lottery scheduling to time multiplex applications across big and little cores. The WASH AMP scheduler [26] classifies threads in applications written in managed languages (e.g., Java) using performance counters and schedules threads to remove bottlenecks (e.g., critical sections). Other works like meeting point thread characterization [47] and X10Ergy [51] propose other means for characterizing and accelerating individual threads on single-ISA heterogeneous platforms. All of these works focus on determining the "critical" task in task-parallel workloads and placing it on the most performant core. Additionally, none deal with cache-incoherent heterogeneous-ISA CPUs, meaning they do not consider data marshaling and cross-node memory access costs.

7 Conclusion

In this work we presented libHetMP, a new OpenMP runtime for efficiently leveraging non-cache-coherent heterogeneous-ISA CPU systems. libHetMP provides the infrastructure necessary for cross-node execution and efficiently distributing parallel computation. libHetMP uses runtime performance measurements as inputs to the novel HetProbe scheduler to automatically make workload distribution decisions, including whether to execute across nodes or only on one node. The HetProbe scheduler, was shown to make sound workload distribution decisions for ten benchmarks and two interconnects. Using the scheduler, libHetMP was able to achieve up to a geometric mean speedup of 41% versus execution solely on Xeon, the best out of the evaluated configurations.

libHetMP shows that it is possible for OpenMP users to take advantage of heterogeneous-ISA CPUs for performance gains. Currently, the interconnect between nodes is the biggest factor determining whether applications can utilize multiple nodes for parallel computation. With cache-coherent interconnects becoming increasingly ubiquitous (e.g., NVLink [40], CXL [17], CCIX [16], Infinity Fabric [3], OpenCAPI [41]), communication will become less of a bottleneck when running across multiple nodes simultaneously. However, even applications that cannot utilize multiple nodes together can select and execute on the node best suited to their application's characteristics. Popcorn Linux and libHetMP together provide a level of execution flexibility not available in other execution contexts (e.g., CPU/GPU systems); there are plenty of avenues for future work in optimizing more irregular applications.

As heterogeneous-ISA architectures become more ubiquitous, it is important that new system software like libHetMP be able to analyze the architecture and automatically adapt application execution to fit. libHetMP provides developers the ability to efficiently leverage emerging heterogeneous CPU systems without extensive manual configuration and deep architectural knowledge.

Our complete implementation is available as open-source as part of the Popcorn Linux project at http://popcornlinux.org/.

Acknowledgement

This work is supported by the US Office of Naval Research (ONR) under grants N00014-16-1-2711, N00014-16-1-2104, and N00014-18-1-2022, and by NAVSEA/NEEC under grant N00174-16-C-0018.

References

[1] PCI Express Base Specification Revision 4.0, Version 1.0, October 2017. https://pcisig.com/specifications/pciexpress/.

[2] Summit: A Supercomputer Suited for AI, June 2018. https://www.olcf.ornl.gov/wp-content/uploads/2018/06/NODE_infographic_FIN.pdf.

[3] AMD. AMD Infinity Architecture Technology, September 2020. https://www.amd.com/en/technologies/infinity-architecture.

[4] Amza, C., Cox, A. L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony,

R., Yu, W., and Zwaenepoel, W. Treadmarks: shared memory computing on networks of workstations. *Computer 29*, 2 (Feb 1996), 18–28.

[5] Anandtech. Intel Agilex: 10nm FPGAs with PCIe 5.0, DDR5, and CXL, April 2019. https://www.anandtech.com/show/14149/intel-agilex-10nm-fpgas-with-pcie-50-ddr5-and-cxl.

[6] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., et al. The NAS parallel benchmarks. *The International Journal of Supercomputing Applications 5*, 3 (1991), 63–73.

[7] Barbalace, A., Lyerly, R., Jelesnianski, C., Carno, A., Chuang, H.-R., Legout, V., and Ravindran, B. Breaking the Boundaries in Heterogeneous-ISA Datacenters. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2017), ASPLOS '17, ACM, pp. 645–659.

[8] Barbalace, A., Sadini, M., Ansary, S., Jelesnianski, C., Ravichandran, A., Kendir, C., Murray, A., and Ravindran, B. Popcorn: Bridging the Programmability Gap in heterogeneous-ISA Platforms. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, ACM, pp. 29:1–29:16.

[9] Barr, J. New – EC2 Instances (A1) Powered by Arm-Based AWS Graviton Processors, November 2018. https://aws.amazon.com/blogs/aws/new-ec2-instances-a1-powered-by-arm-based-aws-graviton-processors/.

[10] Bienia, C. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[11] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 1995), PPOPP '95, ACM, pp. 207–216.

[12] Bueno, J., Planas, J., Duran, A., Badia, R. M., Martorell, X., Ayguadé, E., and Labarta, J. Productive Programming of GPU Clusters with OmpSs. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium* (May 2012), pp. 557–568.

[13] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. 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* (New York, NY, USA, 2005), OOPSLA '05, ACM, pp. 519–538.

[14] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S. H., and Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)* (Oct 2009).

[15] Coarfa, C., Dotsenko, Y., Mellor-Crummey, J., Cantonnet, F., El-Ghazawi, T., Mohanti, A., Yao, Y., and Chavarría-Miranda, D. 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* (New York, NY, USA, 2005), PPoPP '05, ACM, pp. 36–47.

[16] Consortium, C., et al. Cache coherent interconnect for accelerators (ccix). *Online]. http://www. ccixconsortium. com* (2017).

[17] CXL Consortium. Compute Express Link, September 2020. https://www.computeexpresslink.org/.

[18] Daberdaku, S. Parallel computation of voxelised protein surfaces with openmp. In *Proceedings of the 6th International Workshop on Parallelism in Bioinformatics* (New York, NY, USA, 2018), PBio 2018, Association for Computing Machinery, p. 19–29.

[19] DeVuyst, M., Venkat, A., and Tullsen, D. M. Execution Migration in a heterogeneous-ISA Chip Multiprocessor. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 261–272.

[20] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2011), ISCA '11, ACM, pp. 365–376.

[21] Greenhalgh, P. big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. *ARM White paper 17* (2011).

[22] Grewe, D., and O'Boyle, M. F. P. A static task partitioning approach for heterogeneous systems using OpenCL. In *Compiler Construction* (Berlin, Heidelberg, 2011), J. Knoop, Ed., Springer Berlin Heidelberg, pp. 286–305.

[23] Gropp, W., Lusk, E., and Skjellum, A. *Using MPI: portable parallel programming with the message-passing interface*, vol. 1. MIT press, 1999.

[24] Gu, Y., and Mellor-Crummey, J. Dynamic data race detection for openmp programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (2018), SC '18, IEEE Press.

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

[26] Jibaja, I., Cao, T., Blackburn, S. M., and McKinley, K. S. Portable performance on asymmetric multicore processors. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (New York, NY, USA, 2016), CGO '16, ACM, pp. 24–35.

[27] Kale, L. V., and Krishnan, S. *CHARM++: a portable concurrent object oriented system based on C++*, vol. 28. Citeseer, 1993.

[28] Khronos OpenCL Working Group. The OpenCL Specification. Tech. rep., May 2018. https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf.

[29] Kim, J., Jo, G., Jung, J., Kim, J., and Lee, J. A Distributed OpenCL Framework Using Redundant Computation and Data Replication. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2016), PLDI '16, ACM, pp. 553–569.

[30] Kim, S.-H., Lyerly, R., and Olivier, P. Popcorn Linux: Compiler, Operating System and Virtualization Support for Application/Thread Migration in Heterogeneous ISA Environments. Presented at the 2017 Linux Plumbers Conference, September 2017. http://www.linuxplumbersconf.org/2017/ocw/proposals/4719.html.

[31] Kofler, K., Grasso, I., Cosenza, B., and Fahringer, T. An automatic input-sensitive approach for heterogeneous task partitioning. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (New York, NY, USA, 2013), ICS '13, ACM, pp. 149–160.

[32] Kulkarni, M., Burtscher, M., Casçaval, C., and Pingali, K. Lonestar: A suite of parallel irregular programs. In *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software* (2009).

[33] Kumar, A. The New Intel Xeon Processor Scalable Family (Formerly Skylake-SP), August 2017. https://www.hotchips.org/wp-content/uploads/hc_archives/hc29/HC29.22-Tuesday-Pub/HC29.22.90-Server-Pub/HC29.22.930-Xeon-Skylake-sp-Kumar-Intel.pdf.

[34] Lepak, K., Talbot, G., White, S., Beck, N., Naffziger, S., FELLOW, S., et al. The next generation AMD enterprise server product architecture. *IEEE Hot Chips 29* (2017).

[35] Lin, F. X., Wang, Z., and Zhong, L. 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* (New York, NY, USA, 2014), ASPLOS '14, ACM, pp. 285–300.

[36] Luk, C.-K., Hong, S., and Kim, H. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2009), MICRO 42, ACM, pp. 45–55.

[37] Lyerly, R., Kim, S.-H., and Ravindran, B. libMPNode: An OpenMP

Runtime For Parallel Processing Across Incoherent Domains. In *The 10th International Workshop on Programming Modesl and Applications for Multicores and Manycores* (February 2019), PMAM '19.

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

[39] Nelson, J., Holt, B., Myers, B., Briggs, P., Ceze, L., Kahan, S., and Oskin, M. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (Santa Clara, CA, 2015), USENIX Association, pp. 291–305.

[40] NVidia. NVLink, September 2020. https://www.nvidia.com/en-us/data-center/nvlink/.

[41] OpenCAPI Consortium. OpenCAPI Consortium, September 2020. https://opencapi.org/.

[42] OpenMP Architecture Review Board. OpenMP Application Program Interface v5.0. Tech. rep., OpenMP Architecture Review Board, November 2018. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[43] Petrucci, V., Loques, O., and Mossé, D. Lucky scheduling for energy-efficient heterogeneous multi-core systems. In *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems* (Berkeley, CA, USA, 2012), HotPower'12, USENIX Association, pp. 7–7.

[44] Platform, T. N. Next-Generation ThunderX2 ARM Targets Skylake Xeons, 2018. https://www.nextplatform.com/2016/06/03/next-generation-thunderx2-arm-targets-skylake-xeons/.

[45] Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P. Y., and Burger, D. A reconfigurable fabric for accelerating large-scale datacenter services. *Commun. ACM 59*, 11 (Oct. 2016), 114–122.

[46] Qualcomm. Qualcomm snapdragon 855 mobile platform, 2019. https://www.qualcomm.com/media/documents/files/snapdragon-855-mobile-platform-product-brief.pdf.

[47] Rakvic, R., Cai, Q., González, J., Magklis, G., Chaparro, P., and González, A. Thread-management techniques to maximize efficiency in multicore and simultaneous multithreaded microprocessors. *ACM Trans. Archit. Code Optim. 7*, 2 (Oct. 2010), 9:1–9:25.

[48] Ratna, A. A. P., Ibrahim, I., and Purnamasari, P. D. Parallel processing design of latent semantic analysis based essay grading system with openmp. In *Proceedings of the 2017 International Conference on Computer Science and Artificial Intelligence* (New York, NY, USA, 2017), CSAI 2017, Association for Computing Machinery, p. 119–124.

[49] Scogland, T. R. W., Feng, W., Rountree, B., and de Supinski, B. R. CoreTSAR: Core Task-Size Adapting Runtime. *IEEE Transactions on Parallel and Distributed Systems 26*, 11 (Nov 2015), 2970–2983.

[50] Seo, S., Jo, G., and Lee, J. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *2011 IEEE International Symposium on Workload Characterization (IISWC)* (Nov 2011), pp. 137–148.

[51] Shrivastava, R., and Nandivada, V. K. Energy-efficient compilation of irregular task-parallel loops. *ACM Trans. Archit. Code Optim. 14*, 4 (Nov. 2017), 35:1–35:29.

[52] Sutter, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal 30*, 3 (2005), 202–210.

[53] Sutter, H. Welcome to the jungle, August 2012. https://herbsutter.com/welcome-to-the-jungle/.

[54] Venkat, A., and Tullsen, D. M. Harnessing ISA Diversity: Design of a heterogeneous-ISA Chip Multiprocessor. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Piscataway, NJ, USA, 2014), ISCA '14, IEEE Press, pp. 121–132.

[55] von Bank, D. G., Shub, C. M., and Sebesta, R. W. A unified model of pointwise equivalence of procedural computations. *ACM Trans. Program. Lang. Syst. 16*, 6 (Nov. 1994), 1842–1874.