The Effect of Asymmetric Performance on Asynchronous Task Based Runtimes

Debashis Ganguly
Department of Computer Science
University of Pittsburgh
debashis@cs.pitt.edu

John R. Lange
Department of Computer Science
University of Pittsburgh
jacklange@cs.pitt.edu

ABSTRACT

It is generally accepted that future supercomputing workloads will consist of application compositions made up of coupled simulations as well as in-situ analytics. While these components have commonly been deployed using a space-shared configuration to minimize cross-workload interference, it is likely that not all the workload components will require the full processing capacity of the CPU cores they are running on. For instance, an analytics workload often does not need to run continuously and is not generally considered to have the same priority as simulation codes. In a space-shared configuration, this arrangement would lead to wasted resources due to periodically idle CPUs, which are generally unusable by traditional bulk synchronous parallel (BSP) applications. As a result, many have started to reconsider task based runtimes owing to their ability to dynamically utilize available CPU resources. While the dynamic behavior of task-based runtimes had historically been targeted at application induced load imbalances, the same basic situation arises due to the asymmetric performance resulting from time sharing a CPU with other workloads. Many have assumed that task based runtimes would be able to adapt easily to these new environments without significant modifications. In this paper, we present a preliminary set of experiments that measured how well asynchronous task-based runtimes are able to respond to load imbalances caused by the asymmetric performance of time shared CPUs. Our work focuses on a set of experiments using benchmarks running on both Charm++ and HPX-5 in the presence of a competing workload. The results show that while these runtimes are better suited at handling the scenarios than traditional runtimes, they are not yet capable of effectively addressing anything other than a fairly minimal level of CPU contention.

CCS CONCEPTS

• **General and reference** → Evaluation; Performance; • **Software and its engineering** → Multithreading; Multiprocessing / multiprogramming / multitasking; Runtime environments;

This project is made possible by support from the National Science Foundation (NSF) via grant CNS-1421585, and by the 2013 Exascale Operating and Runtime Systems Program under the Office of Advanced Scientific Computing Research in the DOE Office of Science.

KEYWORDS

Runtime Environments, Operating Systems, Performance Evaluation

ACM Reference format:
Debashis Ganguly and John R. Lange. 2017. The Effect of Asymmetric Performance on Asynchronous Task Based Runtimes. In *Proceedings of ROSS '17, Washingon, DC, USA, June 27, 2017,* 8 pages.
DOI: 10.1145/3095770.3095778

1 INTRODUCTION

Task based runtimes are an established class of workloads in the HPC space [10, 14], however they have recently been the subject of a renewed interest as a potential alternative to traditional message passing approaches [8, 13, 21]. To a large extent, this is a result of the growth in complexity and heterogeneity in HPC system architectures, and the sensitivity of traditional message passing models and their resulting bulk synchronous parallel (BSP) execution behaviors. Traditional BSP applications have always been vulnerable to asymmetries in performance due to the reliance on collective based communications. In these situations, the slowest rank of the application determines the overall application performance. Task-based runtimes attempt to alleviate this shortcoming by allowing the runtime much greater flexibility in when and where computation is performed, enabling much more adaptive application behaviors. This flexibility enables the runtime to respond dynamically to load imbalances at the application level, and avoid the straggler problem seen by the more constrained BSP approaches.

Traditionally task based runtimes have focused on load imbalances due to application level behaviors. These imbalances result from asymmetric workloads or generated data being unevenly distributed across the system. To address these issues, task based runtimes adopt an over-decomposition model that allows the rebalancing of small tasks in a fine-grained manner in order to restore the workload balance. This ability to dynamically migrate small components of the workload allow these runtimes to be much more adaptive than traditional BSP models. This adaptability has generated a fair amount of recent interest in these runtimes as it is widely expected that adaptability will be an important capability of future HPC applications due to the increasing variability of the underlying system performance [7, 12, 17, 18]. As a result, developing approaches to cope with performance variability (as opposed to eliminating it) has become a significant challenge for future HPC environments [9, 16, 20].

In addition to addressing performance variability, HPC users are also exploring new methods of composing multiple workload components into complex jobs in order to allow new capabilities such as in-situ analytics and coupled simulations. These compositions generally lead to a number of competing workloads being collocated on the same compute node. While space shared configurations have been the traditional approach for these situations, this has the side effect of potentially underutilizing the node resources as the workloads are often not all computationally intensive or operate in loosely phased manner. As a result, introducing time sharing approaches to these systems is becoming increasingly attractive. The downside to this approach is that timesharing will necessarily introduce asymmetric performance across the node as workloads compete for varying amounts of time on the CPU.

The upshot to this is that consistent and symmetric performance across an HPC system is unlikely to remain a viable expectation. A more realistic assumption is that future extreme scale systems will more closely resemble modern cloud environments than existing petascale class supercomputing systems. While BSP style applications are ill equipped to handle these environments, task based runtimes appear to be ideally suited for them. While asymmetric system performance is different from application level behaviors, from the standpoint of a task based runtime both will manifest as imbalanced load across the node. Therefore, it is generally assumed that the same approaches should be able to address these new challenges.

In this work, we describe an initial evaluation of two task based runtimes executing on a time-shared system as might be expected in future HPC systems. In particular, we investigated the behavior of two modern task based runtimes (Charm++ and HPX-5), when exposed to performance asymmetries caused by the presence of competing workloads. For our evaluation, we modelled an "ideal" time sharing configuration, in that we assumed that (similar to cloud environments) competing workloads were allocated static proportions of the CPU time instead of relying on the default behavior of the scheduler. For our experiments, we leveraged the cgroup features available in current Linux kernels to enforce static resource partitions between the runtime and competing workloads, and then evaluated how effective the runtimes were at utilizing the shared CPU. In our system model, competing workloads do not need to fully utilize a core's processing capacity, and so a certain percentage of the core capacity will be idle. In current space partitioning approaches, these resources are left idle, as there is no way to effectively use them in current applications. The goal of this evaluation was to determine whether task based runtimes could effectively utilize this idle CPU capacity, and so enable the use of time partitioning by leveraging commodity Linux features. The results of our experiments show that, despite their promise, modern asynchronous task based runtimes are not fully capable of responding to asymmetric node performance. Our results show that on average, a modern task based runtime needs at least 75% of a core's computational capacity to effectively utilize it.

2 BACKGROUND

Our work focuses on two popular task-based runtimes: Charm++ and HPX-5. While both Charm++ and HPX-5 provide fine-grained task based processing, each takes widely different approaches to how those tasks are managed. In general Charm++ follows a more centralized load-balancing approach, while HPX-5's approach is based on a decentralized scheme adapted from Cilk [10].

2.1 Charm++

Charm++ [14] is a task-based runtime that implements a migratable-objects programming model. Charm++ is implemented in C++, and leverages the C++ object model in order to provide contained computational objects each of which implements a task in the runtime environment. Because objects are generally self-contained they can easily be migrated across both intra- and inter-node boundaries. The migration mechanism allows Charm++ to dynamically respond to imbalances in the application, made evident by deeper runtime scheduling queues at a particular core or node. Object migration is driven by a dynamic load balancing framework that directs the mapping of particular objects to a given node/core. The primary motivation for load balancing in Charm++ is the propensity for applications to create load imbalance as part of their execution. As an example, an adaptive mesh refinement (AMR) based application will likely see a substantial increase in data at particular locations where the mesh is selected to be refined. This can result in large increase in the amount of work being scheduled at localized locations in the system. In order to mitigate this Charm++ provides a suite of load balancing algorithms that can be selected by the user to balance the load and alleviate any hotspots that it detects. Because applications can have varying causes of load imbalance, Charm++ supports a wide variety of load balancing strategies provided via a configurable set of load balancing implementations, as well as an API to allow the development of more application specific load balancers.

Load balancing in Charm++ is an active process, where a single entity examines the state of the system and actively directs object migration to provide balanced execution across. On a local node, the load balancer runs on a single CPU core (core 0) and examines the scheduler queues of all the other cores on the system. The output of the load balancer is a new assignment of objects to cores that seeks to balance the amount of work included in each object. Because load balancing requires examining the current state of the system the operation causes computation to block across the other cores in the system. As such, the load balancing operation imposes a significant amount of overhead. Due to this, Charm++ offers a set of configuration parameters to determine when and how often a local balancer is invoked. There is also a separate framework (Metabalancer) that automatically invokes the actual load balancer based on runtime heuristics.

2.2 HPX-5

HPX-5 is a task-based runtime based on the Parallax [11] work stealing algorithm inspired by Cilk [10]. HPX-5 provides a set of independent scheduler queues assigned to separate CPU cores. Unlike Charm++ which relies on C++ based objects for its task abstraction, HPX-5 implements a standalone data type called a parcel. Parcels in HPX-5 represent a given computational task and a reference to the data the computation will operate on. In HPX-5, code and data are separate entities and each can be migrated independently of the other. It is thus possible to either move computation to the data or to move data to the computation. This functionality is achieved via a global address space, that allows position independence of the data with respect to the tasks executing in the system.

Unlike Charm++, load balancing in HPX-5 is achieved via hierarchical work stealing conducted by each scheduling entity in the system. If at any point in time, a given CPU core finds that its local scheduler queue is empty (meaning the core is idle), it selects another core and attempts to pull a task from the tail end of its scheduling queue. If there is no work available at the victim, another victim is chosen and the process repeats. The selection of the victim core is random inside a hierarchy level; thus, the first victim core is randomly selected from the same NUMA domain as the stealing core, and only after the local NUMA zone has been checked does the process move up to cores on another socket. What differentiates HPX-5 is the fact that there is no centralized decision-making process to determine how work is balanced across the system. The process is entirely dynamic and done lazily when a given core finds that it has no other work to do. This alleviates any overheads associated with load balancing that result from having to suspend execution while a local balancing operation proceeds.

3 EXPERIMENTAL SETUP

The system used for our evaluation was a dual socket server running two 6-core Intel Xeon E5-2430 processors running at 2.20 GHz. For our experiments, we disabled hyperthreading. The node had 24GB of RAM split evenly across both NUMA zones. The operating system was CentOS release 7.2.1511 with Linux Kernel 3.16.0. All of our experiments were conducted on a single node, as our main goal was to evaluate how a local node's operating system configuration affects the runtimes' performance.

3.1 Benchmarks

The workloads selected for our evaluation were taken from the set of benchmarks published on both the Charm++ and HPX-5 websites. The assumption was that these benchmarks were most indicative of the ideal applications since they were selected to be representative by the runtime projects themselves. For all benchmarks, we ensured that threads were pinned to the cores by the operating system, and other contending workloads were not present in the system.

3.1.1 Charm++. While Charm++ provides a number of different benchmarks [2], we were able to find only one (LeanMD) that supported for load balancing.

- LeanMD: LeanMD simulates the behavior of atoms based on the Lennard-Jones potential. The force calculation in Lennard-Jones dynamics is parallelized using a hybrid scheme of spatial and force decomposition and it mimics the short-range, non-bonded force calculation in NAMD. cutoff-radius, r_c for every simulation space consisting of dimensions that are equal to the r_c and a margin. In each done for all pairs of atoms distance. The force calculation for a to a different set of objects called forces sent by the computes, the cells integration and update various properties of acceleration, velocity and positions. The problem size for our experiments was configured to be $10 * 10 * 10$ dimensions running 101 iterations.

3.1.2 HPX-5. We chose three benchmarks from the HPX-5 applications set [3], which represented a range of different application classes.

- LibPXGL: It represents an implementation of graph algorithms for solving the Single Source Shortest Path (SSSP) problem representing a class of irregular graph problems. chaotic, delta-stepping algorithm. The algorithm distance space into several intervals, where each treated as a bucket. The algorithm proceeds by vertices in each bucket without imposing any processing is done for the current bucket, the progress to the next bucket. Between each bucket there is an explicit barrier on which all localities wait before proceeding to the next bucket. For our experiment, we chose a graph characterized by 2^{20} vertices and $(2^{20} * 16)$ edges. The algorithm runs in chaotic, delta-stepping version with delta size of 4 for 4 problems instances. It should be noted that HPX-5 is designed specifically to handle graph processing applications, so this is one of the target applications for the runtime.
- LULESH: LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) algorithm simulates the Sedov blast wave problem in three dimensions. The problem solution is spherically symmetric and the code solves the problem in a parallelepiped region. We have used the parcels implementation of LULESH which does not take advantage of parallel for loop opportunities and is more directly comparable to the MPI only implementation of LULESH. The problem size for our experiment is defined by 8 domains with 48 points each for 500 iterations.
- HPCG: The current implementation of High Performance Conjugate Gradients (HPCG) is a demonstration of an HPX-5+MPI legacy support modality where HPX-5 and MPI coexist. In this modality, some kernels use HPX-5 whilst others use MPI in order to demonstrate a path forward for porting large legacy applications. For our evaluation, we chose a problem with 32 domains of size $64 * 64 * 64$.

3.2 Competing Workload

Our evaluation used two separate competing workloads. The selection of the workloads was not based on realistic behaviors, but rather to provide the greatest degree of predictability. The goal was to introduce minimal interference other than causing a consistent performance asymmetry in a single CPU core. In effect, we were trying to provide the easiest scenario possible for the runtimes to handle.

- Prime Number Generator: We used a simple utility from the `mathomatic package`. It generates consecutive prime numbers in ascending manner between 0 and 9999999999 using a windowing, memory efficient sieve of Eratosthenes algorithm. Output was redirected to `/dev/null`. This workload is entirely CPU bound and exhibits a minimal memory footprint.
- Kernel Compilation: While not representative of an HPC application, a Kernel compilation does exhibit a large amount of OS level operations. While the prime number generator targets only CPU performance, this workload stresses

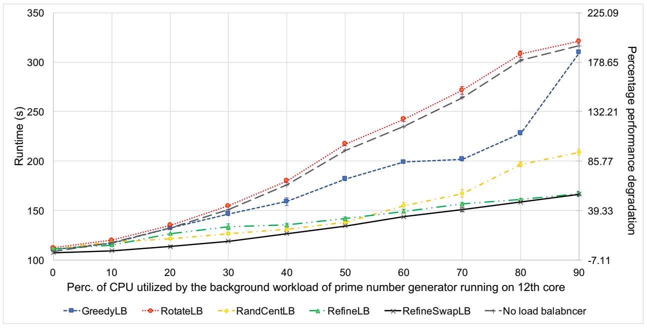


Figure 1: Comparing runtime of LeanMD benchmark in Charm++ configured with different load balancing strategies, and without any load balancer.

internal OS features such as I/O and memory subsystem interference.

For each of these workloads, we confined its execution to a single core in the system. We explicitly avoided using core 0, as it is used by Charm++ to run its load balancers. As well, core 0 tends to be used by other system and runtime features to handle special tasks that might induce more interference if it was contended on. Core assignment was done using the `numactl` utility. The percentage of CPU time available to the competing workloads was enforced by using the `cpulimit` Linux command line tool.

Our experimental configuration is based on assigning each benchmark 12 worker threads running on 12 physical cores, while simultaneously executing a background workload on a single core used by the benchmark. We vary the core utilization of the background workload to estimate the sensitivity of benchmark as measured by its total execution time. We compare against a configuration with 11 worker threads running on 11 cores executing the same benchmark with the same input parameters. However, in the 11 core scenario, the background workload is segregated to the unused 12th core. The two experimental scenarios correspond to a space shared configuration (11 benchmark cores, 1 background core) vs. a time-shared configuration (12 benchmark cores, 1 shared background core). We also compare against the theoretically ideal behavior where the runtime can effectively balance workload without incurring any overhead as a function of the available relative capacity of the total number of CPU cores.

4 CHARM++

The first experiment we conducted was to determine the best load balancer to select for Charm++. Our initial set of experiments were all conducted using the prime number generator as a competing workload. Figure 1 shows the performance of different load balancers in Charm++ as a competing workload is introduced to the system. The experimental environment consists of a single 12 core node running the LeanMD benchmark with each of the available load balancers provided Charm++. Initially, each load balancer produces the similar level of performance, however, they begin to diverge as the competing workload's CPU allocation is increased. The divergence occurs due to the fact that the load imbalance is

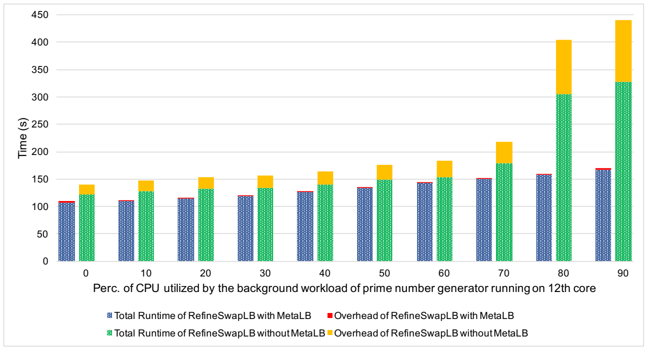


Figure 2: Comparing overhead incurred from load balancing against runtime of LeanMD benchmark in Charm++ for RefineSwapLB with and without MetaLB.

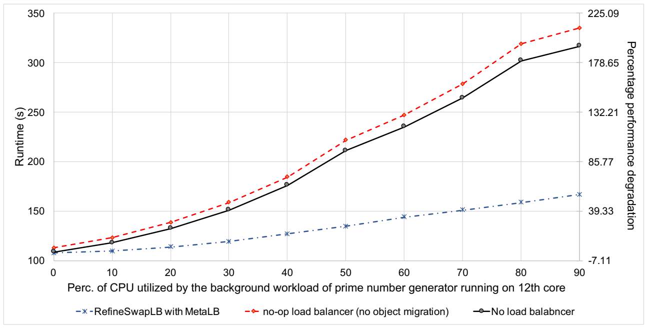


Figure 3: Comparing total job execution time of LeanMD in Charm++ configured with null balancer (no object migration), without any load balancer support, and with the support of RefineSwapLB with MetaLB.

itself increasing as the CPU performance becomes more asymmetrical. With symmetric performance, there is simply no imbalance to correct so the load balancers are effectively doing nothing. As asymmetry increases, however, it is clear that load balancing does help to improve performance (with the exception of RotateLB), and that the RefineSwalLB load balancer improves performance the most. For the remainder of our experiments, we used the RefineSwapLB load balancer.

Our second experiment was meant to determine whether or not to enable the meta-load balancer (MetaLB) in Charm++. Recall that MetaLB implements a dynamic heuristics based trigger to begin load balancing as opposed to a periodic trigger that automatically initiated load balancing after a constant period of time. Figure 2 shows the results of our experiments with and without MetaLB while running the LeanMD benchmark again with varying degrees of competing workloads. The results show that MetaLB consistently delivered better performance across all configurations, primarily due to the fact that it simply did not run the load balancer as often. The majority of the performance gains came from the reduction of the overhead imposed by the load balancer's execution. The rest of our experiments were conducted with MetaLB enabled.

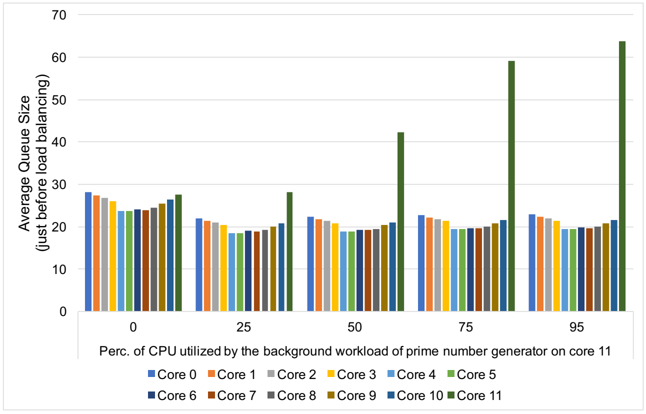


Figure 4: Characterizing the work queues imbalance for different percentages of CPU utilization by the background workload before load balancer redistribute the workload.

Finally, to ensure that our load balancer configuration was, in fact, improving performance for our benchmark in the face of a competing workload we evaluated the execution time of LeanMD with and without load balancing enabled. The results of this experiment are shown in Figure 3. The results show that the load balancer does in fact dramatically improve performance compared to a scenario without load balancing. In addition, this experiment quantified the overheads of the load balancer framework itself, as we implemented our own no-op load balancer implementation that did nothing other than being invoked. The separation between the two top lines in the figure shows the overhead associated with merely invoking the load balancing framework. As can be seen, the performance improvement provided by MetaLB+RefineSwapLB more than makes up for the overhead needed to execute the load balancing framework.

The results here show that (1) load balancing does improve application runtimes in the face of asymmetric CPU core performance, and (2) that the choice of load balancer approach has a significant impact on application performance. Furthermore, as can be seen in Figure 4 asymmetric performance does manifest itself in the same way as application load imbalance since scheduler queue lengths on the core shared with the competing workload increase as the competing workload is ramped up. While it is clear that Charm's load balancers can improve performance on a node with asymmetric performance, the next set of experiments shows whether that performance improvement is actually enough to make it worth it.

Figure 5 shows the overall runtime of the LeanMD benchmark as a competing workload is granted more CPU time on one of the benchmark's core. In this experiment, the application runtime was compared against two other values: (1) the expected performance of the application assuming that the performance loss was proportional to the reduction of CPU time available to the benchmark, and (2) the runtime of the same application configured to use only 11 cores. The expected performance is calculated assuming a baseline with no competing workload (0 on the x-axis) and is calculated as a reduction of the total CPU capacity proportional to the loss of CPU time on one core. So, with a competing workload allocated 50% of

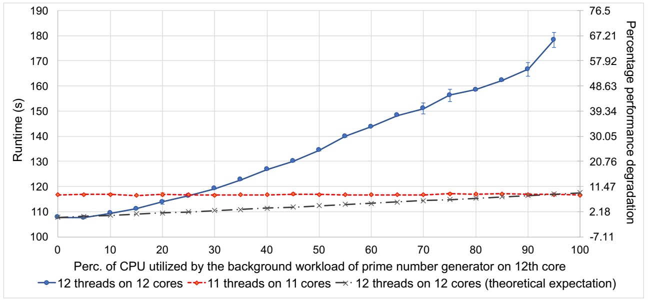


Figure 5: Sensitivity of percentage of CPU utilization by the background workload of prime number generator running on core 11 on the runtime of LeanMD benchmark in Charm++ configured with RefineSwap load balancer coupled with the Meta load balancer.

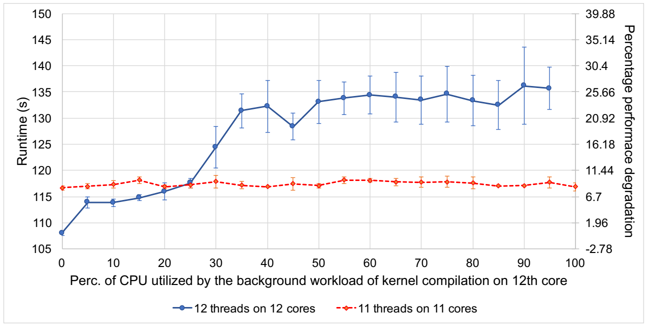
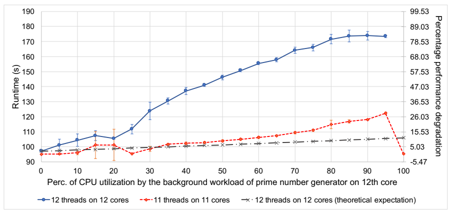


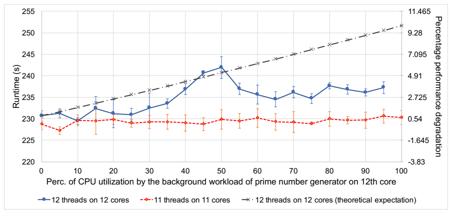
Figure 6: Sensitivity of percentage of CPU utilization by the background workload of kernel compilation running on core 11 on the runtime of LeanMD benchmark in Charm++ configured with RefineSwap load balancer empowered with Meta load balancer.

one CPU the expected performance of a 12 core run would still be 96% of the baseline performance. As can be seen in the figure, the actual results are much worse. In fact, 12 core performance is actually worse than the same application on 11 cores when the competing workload is allocated only 25% of the core capacity. What this means is that if the application cannot get more than 75% of the core's capacity, then it is better off ignoring the core completely. According to our results, a time shared 12 core configuration performs 50% worse than 11 core configuration, even when the time shared layout has 7% more processing capacity. These experiments show that while Charm++ load balancing is effective in reducing the performance penalty due to asymmetric performance, that reduction is generally not enough to make time sharing a viable strategy.

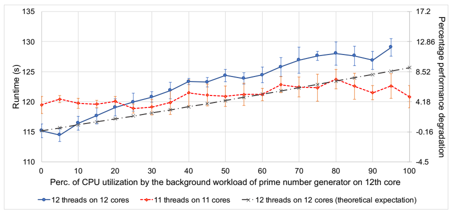
Finally, we evaluated the performance of Charm++ when faced with a more OS intensive workload represented as a kernel compilation. The same set of configuration parameters were used from our earlier experiments. The results are shown in Figure 6. The results show that while the performance is more variant across runs, the high-level behavior is still the same (i.e. if a competing workload uses more than 25% of a core then the core should not be used by the runtime). Another interesting point of the figure



(a) LULESH



(b) HPCG



(c) LibPXGL

Figure 7: Sensitivity of percentage of CPU utilization by the background workload of prime number generator running on core 11 on the runtime of different benchmarks in HPX-5.

is that when compared against prime number generator in the 11 core case the application performance is slightly more variable but overall generally consistent.

5 HPX-5

As described earlier, HPX-5 differs from Charm++ in that HPX-5 uses a work-stealing approach to address load imbalance, as opposed to Charm's active load-balancing approach. As such HPX-5 should avoid many of the overheads associated with Charm's load balancing implementation, and not be reliant on any specific load balancing policy decisions. In order to evaluate HPX-5's ability to address asymmetric performance, we repeated the same set of experiments using the HPX-5 runtime. Due to HPX-5's work stealing approach, we did not include any examination of load balancing overheads.

While the Charm++ results were confined to a single benchmark, with HPX-5 we were able to experiment with a range of different application designs. These benchmarks include two BSP applications that have been adapted to a task based runtime, as well as a graph processing application that is more aptly suited to the HPX-5 environment. Figure 7 shows the experimental results of three HPX-5 benchmarks running with competing prime number generator workload. Just as before we include the 12 core application runtime compared against the expected ideal runtime as well as 11 core configuration that is not sharing a core with the competing workload. Interestingly, both LULESH and HPCG consistently have similar or worse performance with 12 cores when compared to 11 cores, even when there is no competing workload present on the system. In the case of LULESH the performance quickly degrades as the competing workload increases and is 62% worse than the theoretical performance. Furthermore, the 11 core configuration experiences performance degradation as well, but not nearly as severe as the 12 core configuration. With HPCG, the performance degrades the most when the competing workload is granted an equal share of the CPU but otherwise remains generally consistent. In addition, the HPCG performance is actually better than the theoretical performance as the competing workload is granted more CPU time. While LULESH and HPCG are potential outliers due to the fact that they were not originally designed around a task-based programming model, LibPXGL is a much more natural fit to the HPX-5 environment due to its focus on graph processing. Despite that fact, however, LibPXGL still shows the same behavior as was seen with other benchmarks in Charm++. Once the competing workload is

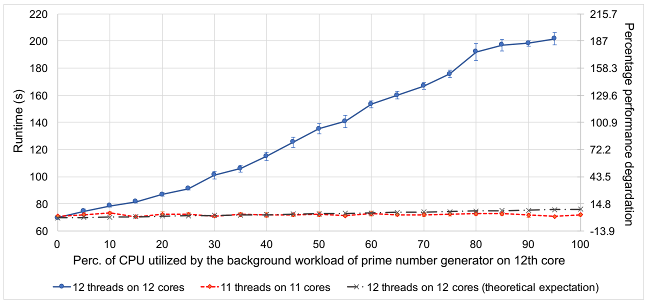
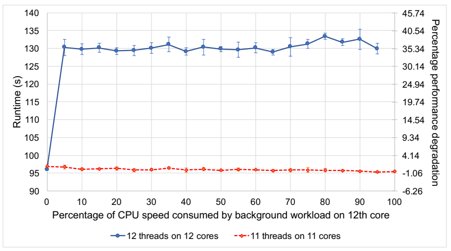


Figure 8: Sensitivity of percentage of CPU utilization by the background workload of prime number generator running on core 11 on the runtime of LULESH benchmark in OpenMP.

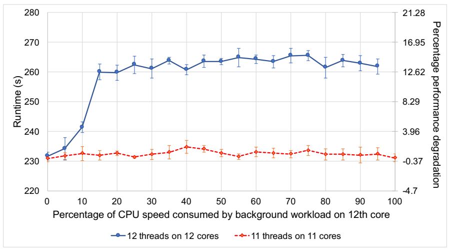
assigned more than 20% of the available CPU time on a single core, the application is better off ignoring the core. In addition, the 11 core configuration shows more performance variance as well in response to the competing workload's configuration changes.

As a point of comparison against other programming models, we also evaluated an implementation of the LULESH benchmark implemented using OpenMP. The purpose was to determine whether or not HPX-5 is better at addressing asymmetric performance than a non-task-based parallelization approach. The results are shown in Figure 8. While the results are generally more consistent, in that the trend-lines are less random than with HPX-5, the performance degradation is substantially worse. Similar to HPX-5, OpenMP shows comparable performance between 11 and 12 cores, when there is no competing workload, however, as a competing workload is introduced, the 12 core runtime increases dramatically. At the same time, the 11 core performance is consistent and even improves slightly as the competing workload is granted more of the CPU. These results show that HPX-5 does, in fact, do better than traditional parallel runtimes, however, it is still not enough to completely overcome the degradation caused by that asymmetry.

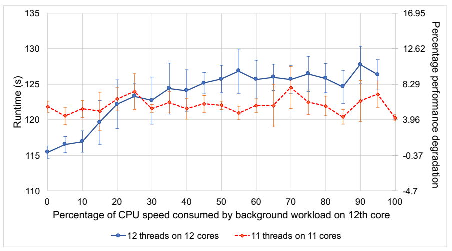
Finally, we reran the experiments with HPX-5 but swapped in the kernel compilation workload as a competing application. The results for these experiments are shown in Figure 9. In general, the results mirror those using the CPU bound prime number generator workload, however, the effects on performance are more immediate. In the case of LULESH and HPCG, instead of showing a



(a) LULESH



(b) HPCG



(c) LibPXGL

Figure 9: Sensitivity of percentage of CPU utilization by the background workload of kernel compilation running on core 11 on the runtime of different benchmarks in HPX-5.

gradual performance decline, the applications instead experience an almost immediate performance hit and maintain the same level of degraded performance even as the competing workload is granted more CPU time. This is most likely due to the inability of Linux's CPU allocation policies to accurately track and assign execution time inside kernel level operations. Since the kernel compilation involves a substantial amount of disk I/O it is likely that the kernel's I/O subsystem is being kept busy and not associating that time to the user level workload correctly. In effect, the CPU allocation loses its meaning as the number of kernel-level operations increase. However, LibPXGL still shows similar behavior to the earlier results. In this case, once the competing workload is granted more than 25% of the core, the 11 core configuration begins to perform better than the 12 core setup. While the performance difference does not continuously increase, the 11 core configuration maintains about 5% advantage in the overall runtime.

6 RELATED WORK

In this work, we focused on two task based runtimes in order to evaluate their ability to handle asymmetric performance. While these two runtimes inhabit fairly different points in the design space there are many other examples that we were not able to evaluate.

Legion [6] is a task-based runtime developed to address heterogeneous processing environments with deep memory hierarchies. Legion focuses on data oriented programming abstractions that tie computational tasks directly to the data regions they will operate on. Legion is similar to Charm++ in that task placement and mapping is done explicitly using a mapping function. Unlike Charm++, however, Legion assumes that the mapping function is application specific and does not support a general framework for use by all applications. Chapel [8] is a new multithreaded programming environment that supports task-based parallelism. Chapel is an entirely new language and runtime that presents high-level abstractions for handling parallel execution as well as data description. The goal of Chapel is to allow users to write very high-level code that is mapped to the hardware by the underlying compiler/runtime. While Chapel does not expose the same level of explicit task based execution as either HPX-5 or Charm++, it does rely on a task based thread execution model.

Qthread [1] is a threading library designed to target highly multithreaded hardware architectures. Its initial design target was the Cray MTA platform which provided large numbers of hardware threads per CPU and a full/empty bit synchronization mechanism in the memory system. While Qthread does not expose an explicit task-based programming model, it does feature a large number of user-level threads with an associated user space scheduling framework and mapping functionality. Similar to Qthread, Argobots [19] is a thread based runtime designed to provide lightweight threads/tasks. Argobots features several new innovations including the concept of Consistency Domains which allows the runtime to operate across non-coherent memory boundaries as well as a stackable scheduling infrastructure allowing the implementation of hierarchical scheduling policies.

The issue of dealing with performance variability and asymmetric performance has been addressed to some degree by the Charm++ community [4, 5]. However, existing work in this area has focused on the issue of performance variability across nodes. In particular, they investigated methods of mitigating the effects of turbo boost and DVFS on Intel platforms, which result in performance inconsistencies at the whole node level. In contrast, this work focuses on the potential intra-node performance asymmetries that can occur due to competing workloads or local OS policy. The existing work in Charm++ explicitly disregards this configuration and ensures that local node performance is kept as consistent as possible. Finally, while Charm++ takes an active approach to managing load imbalances, there is existing work that integrates hierarchical greedy work stealing strategies into the runtime [15].

7 CONCLUSION

In this work, we examined the ability of modern task based runtimes to handle asymmetric performance on a local compute node. In future HPC environments, it is expected that workloads will increasingly be consolidated onto the single compute nodes, necessitating new approaches to managing the system software environments. As HPC systems begin to resemble cloud systems, it is worth considering whether cloud system software approaches are adaptable to the HPC space. In this work, we explored how well task based runtimes reacted to system environments based on existing cloud infrastructures. Based on our results, we have shown that while modern task based runtimes are able to better manage a system with asymmetric CPU performance, they are still unable to effectively make use of time-shared resources. As our experiments show, on average a CPU loses its utility to a task based runtime as soon as its performance diverges by only 25%. In other words, if a CPU is less than 75% idle, a modern task based runtime will be unable to effectively use its available computational capacity. These results are consistent across both Charm++ and HPX-5, which both exhibit widely different approaches to addressing the performance asymmetry and the resulting load imbalances that it causes. Based on these results, we believe that in future work consistent and low noise HPC operating systems remains a necessary endeavor to support future HPC applications.

REFERENCES

[1] 2014. The Qthread Library. http://www.cs.sandia.gov/qthreads/. (2014). Accessed April 5.
[2] 2017. Charm++ Mini-apps. http://charmplusplus.org/benchmarks/. (2017). Accessed April 5.
[3] 2017. HPX-5 Applications. https://hpx.crest.iu.edu/applications. (2017). Accessed April 5.
[4] Bilge Acun and Laxmikant V Kale. 2016. Mitigating Processor Variation through Dynamic Load Balancing. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 1073–1076.
[5] Bilge Acun, Phil Miller, and Laxmikant V Kale. 2016. Variation among processors under turbo boost in hpc systems. In *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 6.
[6] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*. IEEE Computer Society Press, 66.
[7] Abhinav Bhatele, Kathryn Mohror, Steven H Langer, and Katherine E Isaacs. 2013. There goes the neighborhood: performance degradation due to nearby jobs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 41.
[8] Bradford L Chamberlain, David Callahan, and Hans P Zima. 2007. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.
[9] Matthieu Dorier, Gabriel Antoniu, Franck Cappello, Marc Snir, Robert Sisneros, Orcun Yildiz, Shadi Ibrahim, Tom Peterka, and Leigh Orf. 2016. Damaris: Addressing Performance Variability in Data Management for Post-Petascale Simulations. *ACM Transactions on Parallel Computing (TOPC)* 3, 3 (2016), 15.
[10] Matteo Frigo, Charles E Leiserson, and Keith H Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *ACM Sigplan Notices*, Vol. 33. ACM, 212–223.
[11] Guang R Gao, Thomas Sterling, Rick Stevens, Mark Hereld, and Weirong Zhu. 2007. Parallex: A study of a new parallel computation model. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, 1–6.
[12] Yuichi Inadomi, Tapasya Patki, Koji Inoue, Mutsumi Aoyagi, Barry Rountree, Martin Schulz, David Lowenthal, Yasutaka Wada, Keiichiro Fukazawa, Masatsugu Ueda, and others. 2015. Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 78.
[13] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 6.
[14] Laxmikant V Kale and Sanjeev Krishnan. 1993. CHARM++: a portable concurrent object oriented system based on C++. In *ACM Sigplan Notices*, Vol. 28. ACM, 91–108.
[15] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V Kale. 2012. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*. ACM, 137–148.
[16] Aniruddha Marathe, Peter E Bailey, David K Lowenthal, Barry Rountree, Martin Schulz, and Bronis R de Supinski. 2015. A run-time system for power-constrained HPC applications. In *International Conference on High Performance Computing*. Springer, 394–408.
[17] Oscar H Mondragon, Patrick G Bridges, Scott Levy, Kurt B Ferreira, and Patrick Widener. 2016. Understanding performance interference in next-generation HPC systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 33.
[18] Allan Porterfield, Rob Fowler, Sridutt Bhalachandra, Barry Rountree, Diptorup Deb, and Rob Lewis. 2015. Application runtime variability and power optimization for exascale computers. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 3.
[19] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, P Beckman, C Bordage, G Bosilca, A Brooks, A CastellAs, D Genet, T Herault, and others. 2015. Argobots: A lightweight low-level threading/tasking framework. (2015).
[20] Akshay Venkatesh, Abhinav Vishnu, Khaled Hamidouche, Nathan Tallent, Dhabaleswar DK Panda, Darren Kerbyson, and Adolfy Hoisie. 2015. A case for application-oblivious energy-efficient MPI runtime. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 29.
[21] Jeremiah J Wilke. 2015. *Dharma: Distributed asynchronous adaptive resilient management of applications.* Technical Report. Sandia National Laboratories (SNL-CA), Livermore, CA (United States).