2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

Multi-level Load Balancing with an Integrated Runtime Approach

Seonmyeong Bak*, Harshitha Menon†, Sam White*, Matthias Diener* and Laxmikant Kale*
*University of Illinois at Urbana-Champaign
Email: {sbak5,white67,mdiener,kale}@illinois.edu
†Lawrence Livermore National Laboratory
Email: harshitha@llnl.gov

***Abstract*—The recent trend of increasing numbers of cores per chip has resulted in vast amounts of on-node parallelism. These high core counts result in hardware variability that introduces imbalance. Applications are also becoming more complex, resulting in dynamic load imbalance. Load imbalance of any kind can result in loss of performance and system utilization. We address the challenge of handling both transient and persistent load imbalances while maintaining locality with low overhead.**

In this paper, we propose an integrated runtime system that combines the Charm++ distributed programming model with concurrent tasks to mitigate load imbalances within and across shared memory address spaces. It utilizes a periodic assignment of work to cores based on load measurement, in combination with user created tasks to handle load imbalance. We integrate OpenMP with Charm++ to enable creation of potential tasks via OpenMP's parallel loop construct. This is also available to MPI applications through the Adaptive MPI implementation. We demonstrate the benefits of our work on three applications. We show improvements of *Lassen* by 29.6% on Cori and 46.5% on Theta. We also demonstrate the benefits on a Charm++ application, *ChaNGa* by 25.7% on Theta, as well as an MPI proxy application, *Kripke*, using Adaptive MPI.

***Keywords*-Charm++, OpenMP, Adaptive MPI, Load Balancing, Hybrid Programming**

I. Introduction

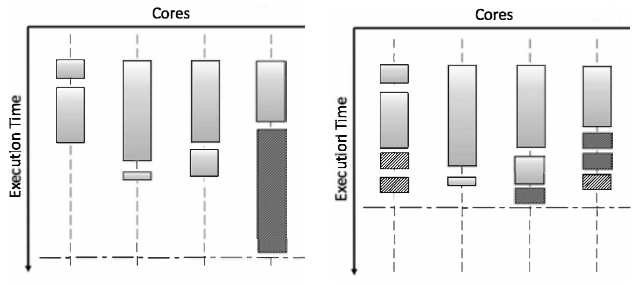
Several trends in high-performance computing are converging to drive applications and systems software to rely on multi-threading in each node's shared memory, rather than running an independent process on each CPU core. The general abandonment of specialized OS kernels [1], [2] in favor of general-purpose Linux has rolled back past efforts to reduce noise from system processes [3]. Finally, CPU heterogeneity [4] and increasing application sophistication both increase load imbalance and unpredictability.

In this paper, we present a combination of the Charm++ and Adaptive MPI [5] distributed programming models with OpenMP that addresses many of these challenging trends with a low-overhead and locality-conscious design.

The number of cores and hardware threads in each chip is increasing rapidly. Within each node, increased hardware parallelism entails reduced per-core/thread memory capacity and bandwidth. Over entire parallel systems, treating each core as an independent unit forces communication libraries to consume more memory and pushes collective algorithms further toward asymptotic scaling limits. Applications that wish to use each core independently must be structured to expose a correspondingly large and growing degree of parallelism. General whole-job load balancing mechanisms must then address the increased scale of both systems and applications. Thus, the prospect of grouping many cores together as multi-threaded units mitigates many threats to continued performance scaling.

Many parallel applications no longer operate in a regime where work and data can be neatly divided into uniform chunks distributed to each processor. This trend encompasses unstructured computations, data-dependent iterative methods, variable resolution, multi-physics simulations, multi-phase execution, and many other developments that trade reduced total work or increased accuracy for more complicated and less predictable execution. Even applications that do offer simple structured decompositions can be made imbalanced by hardware heterogeneity. Load balancing in various forms can be applied to aid these applications, but it too must be scalable, which often means coarsening the problem to the node level to avoid considering an excessive number of cores. Discrete units of work assignment, heuristic algorithms, and unpredictable processor performance also prevent perfect uniformity. Supplementary within-node balancing can help make up for these short-falls, as illustrated in Figure 1. As shown in the figure, the initial distribution of work items, represented by blocks, results in load imbalance. Using supplementary within-node load balancing helps redistribute only the excess work, as represented by the shaded blocks from the overloaded cores.

Even with very balanced work assignment across nodes and individual cores, execution may not proceed at a perfectly uni-



(a) Without intra-node LB. (b) With intra-node LB.

Fig. 1: The potential benefits of intra-node work sharing on reducing load imbalance.

form pace. Network contention can delay some messages more than others. System noise from OS processes can also non-uniformly interfere with execution [3], with hard to predict knock-on effects [6]. Dynamic work redistribution can greatly help in mitigating these effects [7].

All of these pressures lead to a conclusion that multiple cores within each node must share data and work to sustain continued scalability in problem size and performance. At the same time, any sharing mechanism ideally should not compromise data locality or introduce excessive new bottlenecks or overheads. To address these desires, we introduce a design that combines the Charm++ and Adaptive MPI distributed programming models with a modified OpenMP runtime system. Charm++ intermittently performs coarse load balancing in terms of objects that encapsulate associated work and data together, and assigns them to particular cores with good balance among nodes. These objects then adaptively share work with other cores in the same process, exposing fine-grained tasks only to the extent that otherwise idle cores are available to help execute them. Thus, our design ensures locality as well as low and proportionate scheduling overhead. We demonstrate this design's effectiveness through the improved performance and scalability of several applications on large supercomputers.

The contributions of this paper are:

- An approach that combines infrequent distributed load balancing with shared-memory task parallelism to handle persistent and transient load imbalances together.
- Efficient implementation of dynamic scheduling of fine-grained tasks which uses an adaptive schedule based on the state of the system.
- Integration of OpenMP with Charm++'s runtime system to enable fine-grained parallelism.
- Improved performance by using the integrated runtime system on three different applications. We show improvements of 29.6% on Cori and 46.5% on Theta with Lassen and 25.7% on Cori with ChaNGa. We also show the benefit on a MPI application, Kripke, using Adaptive MPI (AMPI).

II. Related Work

Per-chip core and thread counts are steadily increasing in HPC systems. The trend toward increasing core/thread counts is accelerating with the increased deployment of Knight's Landing-generation Intel Xeon Phi hardware with several dozen cores per chip as primary processors rather than as accelerators (e.g. in NERSC's Cori, LANL's Trinity, and ANL's Theta). This trend has driven scalability challenges and opportunities for increased efficiency arising from multiple cores sharing access to common memory. MPI has correspondingly evolved in usage and implementation to work well in this setting [8], [9], [10], [11], [12], [13], leading to explicit support for shared memory in the MPI-3 standard. Charm++ has followed a similar progression, as described in Section III.

The process-per-core model of pure MPI has not been universally sufficient. Applications may have limitations in the scalability of their parallel algorithms and data structures, or may present insufficient parallelism in their mode of work decomposition among MPI processes. Communication that could be avoided in shared memory is also an undesirable overhead. This has led to the rise of hybrid 'MPI+X' programming. OpenMP is the most prevalent shared-memory programming model paired with MPI, with extensive work studying its implementation and impact (e.g. [14], [15], [16]). This hybrid model has been increasingly used with other shared memory programming models to handle within node parallelism [14], [17], [18].

The MPI+X model itself can improve load balance within each node [19]. We combine a periodic measurement-based inter-node load balancing scheme to attain approximate uniformity, with dynamic shared-memory execution to smooth out residual imbalances. Recent papers explored the hybrid model in more detail [20], [21], [22]. They mix static and dynamic scheduling of work among cores on a node to improve the tradeoffs among overhead, locality, and load imbalance. They also show that these techniques can be used to reduce the impact of system noise [7]. Our work carries these ideas further, by adaptively tuning the level of dynamic scheduling to match its potential utility, thus reducing overhead.

Projects to more tightly integrate various shared and distributed memory models have also arisen, with aims to improve scheduling and locality further. OmpSs [17] introduced concurrent tasks on top of OpenMP, with data dependences satisfied by MPI communication operations and coordinated by its runtime system. Recent versions of MPC bind an implementation of MPI that supports multiple ranks in each OS process [23] to multi-threading via POSIX threads [24], OpenMP [25], and Intel TBB. This paper moves in a similar direction, by directly scheduling execution of various shared-memory tasks to run on normal Charm++ worker threads, overlaid on the work and data mappings generated by Charm++'s distributed memory load balancing infrastructure.

The approach of work-stealing task scheduling has been used in Cilk [26], Intel TBB [27], OpenMP 3.0 [28] and Habanero [29]. The randomized work-stealing used in Cilk can result in loss of locality. TBB has a mechanism to bind each loop iteration to the same worker thread that previously executed that iteration, thereby favoring temporal cache-reuse. The Habanero runtime system has an adaptive locality-aware work-stealing scheduler [30] to increase temporal data reuse.

III. The Charm++ Programming Model

Charm++ is a parallel programming system which is based on an asynchronous message driven execution model. Each application's data and computations are encapsulated in entities called *chares*, which are C++ objects. An application written in Charm++ is over-decomposed into these objects. Chares interact via asynchronous method invocations and a method on a chare is executed when a message is received for it. Chare objects are assigned to a core by the runtime system. Typically there are many more objects than the number of cores, which is known as over-decomposition. This encapsulation of data

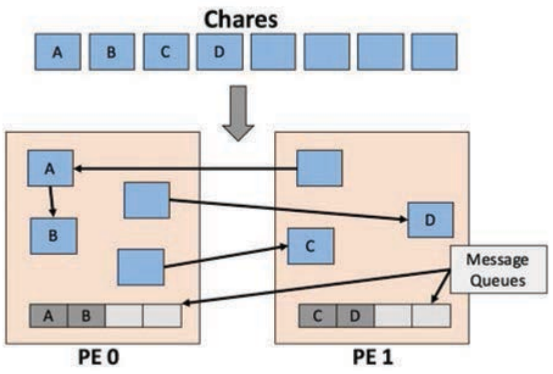


Fig. 2: The Charm++ parallel programming system.

and its computation into a chare, each of which is mapped to a specific core, inherently promotes data locality.

In the message driven execution model of Charm++, the runtime system actively probes for incoming messages. On receiving a message, it identifies the corresponding *chare* which is targeted by the incoming message and schedules it. Figure 2 shows the overdecomposition where multiple *chares* are assigned to a PE and are communicating via messages. A *PE* here refers to a processing element such as a *core* or a *hardware thread*, and we use these terms interchangeably.

The *SMP mode* of Charm++ takes advantage of multi-core shared memory processors [31]. In this mode, a Charm++ OS process is called an SMP node which launches multiple threads. Each thread is called a PE. In a typical configuration, the number of threads launched by the Charm++ process is equal to the number of cores or hardware threads on a node. A PE is mapped to a separate core or a hardware thread. PEs have CPU affinity, that is, each PE is bound to a core and the operating system is not allowed to migrate it to another core. Each PE has a separate message queue and the scheduler on the PE picks up messages from the queue and handles it. Within an SMP node, data is shared between PEs via pointers.

Utilizing the multi-core processors in this way has many benefits. In SMP mode, intra-node communication is implemented via a single copy, rather than the double copy scheme used between nodes. It also significantly reduces the memory footprint of the program by eliminating the memory needed for intra-node communication channels and buffers. Since all PEs within an SMP node share a memory address space there needs to be only one copy of read-only data structures. Running multiple threads in a single process enables work sharing without explicit inter-process data transfer.

IV. Overview of our Proposal

The challenge, as outlined in Section I, is to balance load across PEs while managing locality. A pure task model with randomized work stealing, or a pure dynamic schedule in OpenMP, sacrifices locality significantly to an extent that often eliminates the benefits of dynamic load balancing [20], [22]. Dynamic load balancing strategies are used to balance the load and redistribute the work at runtime. These load balancing strategies can incur significant overhead due to the cost of computing a new assignment and the consequent data movement. If done less frequently, the overhead is reduced and locality is maintained, but dynamically emerging load imbalance may last longer before being corrected. With increasing number of cores within a node, intra-node load balancing will become an effective way to reduce load imbalance.

The approach we propose is to utilize a relatively infrequent periodic assignment of work to cores based on load measurement, combined with user assisted creation of potential tasks from the work assigned to each core that the runtime can choose to make available to other cores. The idea is to utilize the idle cycles on other cores on a node to execute tasks belonging to the overloaded cores. We also need to make sure we do not incur task creation overhead when tasks are not needed. Figure 1 shows a schematic diagram of such a scenario where most of the computations are executed on the core they are assigned to, but the load imbalance towards the end triggers the dynamic creation of fine-grained tasks which are distributed across different cores.

We support this approach with a method for users to create potential tasks. This method builds on top of a new task abstraction in Charm++ and integrates OpenMP with Charm++, such that each object can create potential tasks via OpenMP parallel loop constructs. Using this method, a user can create potential tasks that can dynamically utilize all cores to restore balance. We also develop multiple runtime scheduling strategies for managing these potential tasks.

In the following sections we describe our approach in detail. We first discuss periodic distributed load balancing in Section V. Following this we describe our OpenMP integration with Charm++ in Section VI. Finally, we showcase the application performance improvements achieved by using the new integrated runtime system in Section VII.

V. Persistence-based Load Balancing

Many HPC applications execute the simulation in a series of time-steps or iterations until convergence is achieved. As a result, consecutive iterations have a similar computation and communication pattern. For such applications, a heuristic called *principle of persistence* [32] holds which says that the communication pattern and computation load of the recent past is a good indicator of the near future. We use this to predict the load of future iterations; the predictions are used by plug-in load balancing strategies to make the global decisions. We work with Charm++ because of its support for dynamic load balancing. As mentioned earlier, in Charm++, the data and its computation are encapsulated into a chare object which resides on a specific processing element (PE). This naturally promotes locality. Load balancing aims to provide an assignment of these objects to PEs to reduce the load imbalance. The Charm++ load balancing framework provides a mechanism to collect the load and communication statistics of each chare object and the processor in a distributed database. These statistics are used by the load balancing strategies to generate a chare-to-core mapping at run time.

Charm++ contains a suite of load balancing strategies that balances load between PEs. For the purpose of this work, we use a hierarchical hybrid load balancing strategy for one of the applications, Lassen. In this hierarchical strategy, the processors are divided into groups organized in a hierarchical tree fashion. At each level of the hierarchy, the root performs the load balancing strategy over the children in its sub-tree. The residual load imbalance that results in spite of this periodic balancing can be handled by the fine-grained intra-node task balancing strategies described below.

VI. OpenMP Interoperation with Charm++

In this section, we discuss the OpenMP thread model and our implementation and optimization of its runtime features for Charm++. We introduce the prior implementation of the integration and discuss the changes we made over the prior implementation afterwards.

A. Overview and limitations of the prior OpenMP integration

Initially, we implemented this integrated runtime using GNU OpenMP [33], [34]. In the first implementation, we used stackless Charm++ messages to implement OpenMP threads on top of Charm++ runtime. Each chare can create OpenMP threads, which become stackless Charm++ messages which can be stolen across PEs within the same node. These OpenMP threads are pushed to a PE-local work-stealing queue, which is implemented using the Chase-Lev [35] non-blocking algorithm. To minimize the overhead, we adopted two heuristics. Each node maintains an atomic counter to keep track of idle PEs within a node and each PE keeps a history vector of how many OpenMP tasks have been stolen by other idle PEs. Using these two heuristics, we can create OpenMP tasks only when there are idle PEs and the fine-grained parallelism is beneficial.

The initial implementation still has a creation overhead to some degree and has only limited support for OpenMP directives, because it is implemented using stackless Charm++ messages. First, it only supports barriers at the end of each OpenMP parallel region. OpenMP has implicit and explicit barriers within a region, and can use multiple barriers within each region. For example, 'omp for' has an implicit barrier in the end of each 'omp for' pragma and 'omp single' may have an implicit barrier if the variable updated within 'omp single' is accessed outside the pragma. In addition, many synchronization pragmas such as 'omp barrier' are used for correctness and verification. These barriers could not be implemented because stackless messages were used.

To implement barriers, the OpenMP tasks should be able to be suspended and resumed, and all the data for each OpenMP task should be maintained when they are resumed on other PEs. In addition, the stackless messages incur unnecessary overhead for each OpenMP parallel region. Most OpenMP runtimes maintain a pool of threads which are suspended and can be resumed for the upcoming OpenMP parallel regions, such that an OpenMP thread is initialized only when it is created in the beginning of the first OpenMP parallel region, and is suspended and resumed until the runtime is exiting. Our initial implementation did the initialization for each OpenMP parallel region because threads could not be suspended and resumed.

B. Overview of the current implementation

We adopted user-level threads to resume and suspend OpenMP tasks on top of the Charm++ runtime. Now, each OpenMP parallel region creates user-level threads which can be scheduled by the Charm++ runtime scheduler. These user-level threads are pushed to the same work-stealing queue as in the first queue and minimize the overhead of fine-grained parallelism using the same heuristics in the prior implementation. We use boost context assembly codes to implement migration of user level threads across different kernel threads in Charm++ runtime system. Each user level thread has its own stack and is migratable across different kernel-level threads.

However, even with the user-level threads, there are still several issues to implement suspend and resume of OpenMP threads. The first issue is how to schedule suspended OpenMP tasks which are stolen by thieves. Thieves cannot continue to work on this because they can be idle temporarily while waiting for messages from other PEs. So, these suspended tasks should be pushed to the creator's queue. The second issue is that the suspended tasks cannot be pushed to the creator's work-stealing queue by thieves because the work-stealing queue supports one producer and multiple consumers to minimize the usage of atomic operations. To resolve this issue, we implemented a separate queue for suspended tasks on each PE which supports multiple producers and consumers.

Figure 3 shows how the current implementation of OpenMP interoperates with Charm++ on a node with 2 PEs. First, the integrated OpenMP creates OpenMP tasks on OpenMP parallel region which are user-level threads migratable across PEs in Charm++. Each OpenMP parallel region keeps an atomic counter for each barrier within the parallel region. Created OpenMP tasks decrement the counter when encountering barriers within each OpenMP parallel region and they are pushed to the creator's suspended task queue if they are executed on PEs other than the creator. The creator waits for the counter to become zero and moves suspended tasks from the suspended task queue to the work stealing queue afterwards. In this way, the integrated OpenMP resolves load imbalance across PEs within a node and implements synchronization and worksharing directives of OpenMP on top of the Charm++ runtime.

We initially modified the GNU OpenMP runtime for our work but we migrated to LLVM OpenMP runtime for better compatibility which works with common compilers such as icc, gcc, and clang. In addition to better compatibility, the LLVM OpenMP runtime has fine-grained optimizations such as frequent usage of padding for shared variables and assembly instructions for synchronization routines.

C. Benefit of the current version over the initial version

The adoption of user level threads brings several advantages over the initial implementation. First, multiple OpenMP par-

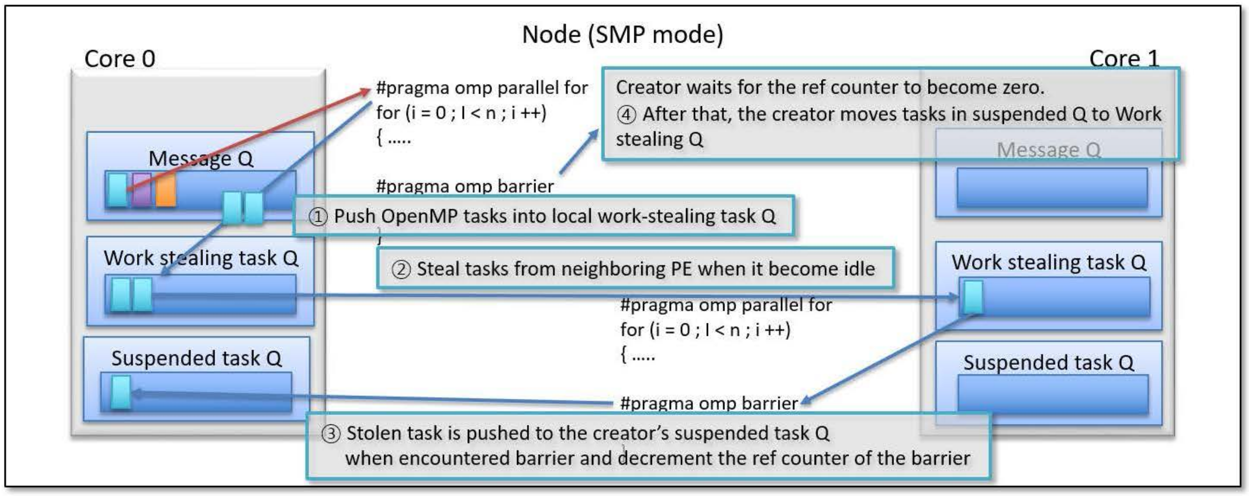

Node (SMP mode)
Core 0
Core 1
#pragma omp parallel for
for (i = 0 ; I < n ; i ++)
{ .....
#pragma omp barrier
}
Creator waits for the ref counter to become zero.
④ After that, the creator moves tasks in suspended Q to Work stealing Q
Message Q
Message Q
① Push OpenMP tasks into local work-stealing task Q
Work stealing task Q
② Steal tasks from neighboring PE when it become idle
Work stealing task Q
#pragma omp parallel for
for (i = 0 ; I < n ; i ++)
{ .....
Suspended task Q
Suspended task Q
#pragma omp barrier
③ Stolen task is pushed to the creator's suspended task Q
when encountered barrier and decrement the ref counter of the barrier

Fig. 3: Implementation of OpenMP for Charm++ using user-level threads.

allel parallel regions can be merged into one bigger OpenMP parallel region. At the start of an OpenMP parallel region, the runtime incurs an overhead and loses locality if there are short, successive OpenMP parallel regions because the same data can be accessed by different PEs. Implementation of barriers resolves this issue by merging short OpenMP parallel regions into a bigger parallel region. In addition, we can avoid some of the initialization of each OpenMP task mentioned above because tasks can suspend and resume within a while loop. Each PE keeps a pool of user-level threads for OpenMP and resumes those threads only with initialization of function pointers to each OpenMP parallel region.

VII. Application Studies

We study the performance benefits of our new integrated runtime system that combines the Charm++ distributed memory model with the task model on 4 different applications. First, we studied the characteristics and limitations of periodic load balancing with Lassen. And show the benefit of this integrated runtime on the applications. We choose 2 Charm++ applications and 1 MPI+OpenMP applications such as Lassen, ChaNGa, and Kripke. We compare the performance of these codes with and without the task model integrated. We show the performance of all applications on NERSC Cori and ALCF Theta. For all the applications, we picked the scheduling strategy that performed the best. We use two heuristics for OpenMP to minimize overhead.

Cori and Theta are CrayXC machines hosted by Lawrence Berkeley National Laboratory and Argonne National Laboratory. Cori has two different kinds of nodes, Haswell and KNL. For Haswell nodes, Cori has two 16-core Intel Xeon E5-2698 v3 processors on each node. Theta has only KNL nodes which consists of Intel Xeon Phi 7230. We used Haswell nodes on Cori and KNL nodes on Theta for experiments.

A. Lassen

Lassen is an LLNL proxy application for modeling wave propagation by tracking the wave front. This application has significant load imbalance where the load is concentrated just before and after the wave front. As the wave front moves, computation load also shifts. We use the Charm++ implementation of Lassen. The input to the application is a Cartesian mesh subdivided into domains and assigned to PEs. The number of domains used is sixteen times the number of PEs. We use 2-way SMT on both Cori and Theta for Lassen, which have 32 and 64 cores per node. First, we run Lassen on a single node of Cori with different load balancing schemes, different frequency of LBs and decomposition ratios to illustrate the limitations of periodic load balancing. In these experiments, we measure the load imbalance of Lassen with different load balancers and calculate the *percent imbalance* λ [36] with Equation 1. In the equation, L is the load vector. A higher value of λ indicates a higher imbalance.

$$\lambda = \left(\frac{max(L)}{avg(L)} - 1\right) \times 100\% \tag{1}$$

We use four load balancing strategies: GreedyLB, GredyRefineLB, RefineLB and HybridLB. GreedyLB moves objects from most loaded PEs to least loaded PEs. RefineLB computes the average loads across PEs and move objects so that loads on each PE get closer to the average. GreedyRefineLB works similar to GreedyLB but minimizes migration of objects.

HybridLB combines different load balancing strategies in multiple levels of hierarchy of PEs. In HybridLB, the root collects load measurements from all the PEs and makes decisions on migration of objects in the first level of the hierarchy with a predefined load balancing strategy for the level. PEs in the first level migrate objects across siblings based on the decision in the first level, then become the root for each region of PEs in the second level and perform the same procedure. This hierarchical load balancing can minimize migration of objects between PEs which are located far from each other and reduce the overhead of centralized load balancing. In addition, we can use different LBs that work better in each level. We used HybridLB with two levels and adopted RefineLB and GreedyRefineLB for the first and second level, respectively.

Figure 4 shows the load imbalance factor λ with different

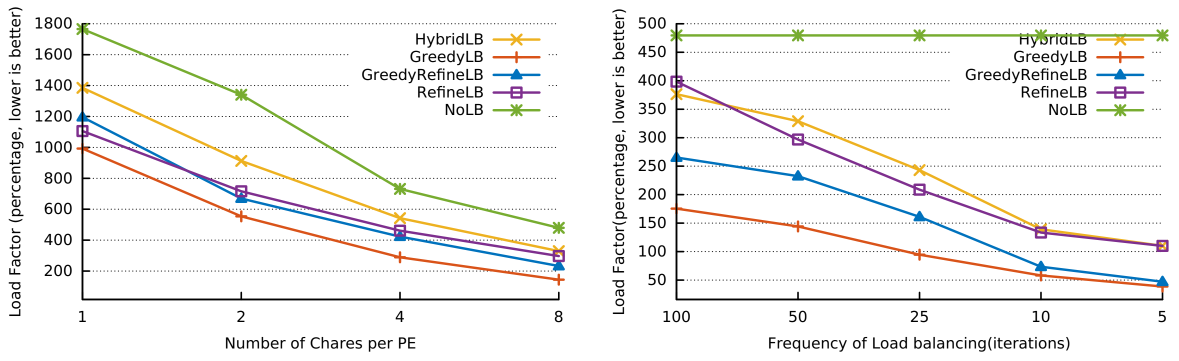


Fig. 4: Load imbalance factor λ of Lassen with different configurations of decomposition and load balancing on a single node of Cori without OpenMP integration.

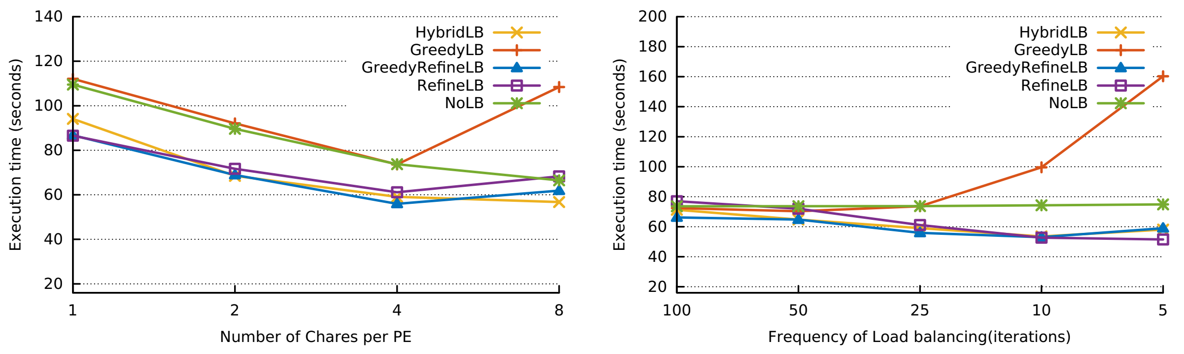


Fig. 5: Performance of Lassen with different configurations of decomposition and load balancing on a single node of Cori without OpenMP integration.

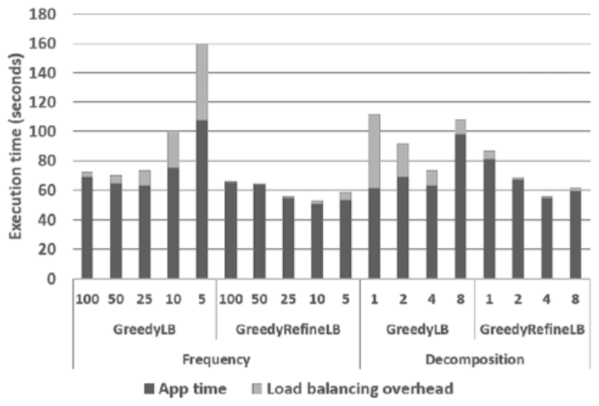
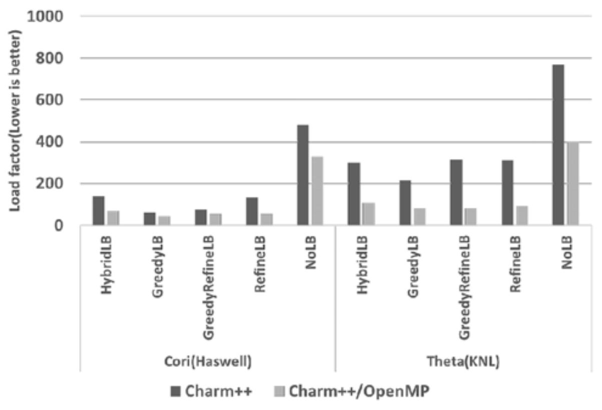


Fig. 6: Application time and Load balancing overhead of Lassen with GreedyLB and GreedyRefineLB on a single node of Cori without OpenMP integration.

decomposition ratios and load balancing configurations on Cori. Load imbalance is reduced by higher ratio of decomposition and frequency of load balancing. However, the performance of Lassen get worse or does not improve from certain decomposition ratios and frequencies. Figure 5 shows the performance of Lassen with different decomposition ratio and frequency of LB. Lassen shows the best performance with 4 chares per PE and 10 iterations with GreedyRefineLB and RefineLB on a single node. GreedyLB doesn't work well even compared to execution runs without load balancing. This performance degradation of decomposition and load balancing comes from incurred overhead. As we decompose problem domain into more objects, the surface to volume ratio increases, which means application will spend more time on communication between neighboring objects in the problem domain. As we increase the frequency of LBs, the application should do some global communications to collect load measurement and migration of objects across PEs and nodes. Figure 6 shows the detailed timing result of GreedyLB and GreedyRefineLB in stacked bar graph of application time and load balancing overhead. The load balancing overhead includes cost of migrating objects and global synchronization cost for collecting load measurements on each PE. GreedyLB shows increasing load balancing overhead while GreedyRefine maintains marginally increasing overhead because GreedyLB doesn't consider the assignment of objects, which incurs more migrations. GreedyRefine minimizes the migration of objects considering the assignment of objects. In addition, this load balancing overhead affects the application time because each PE continues its work just after they finish their contribution and migration for each load balancing. So, while other PEs migrating their objects, some PEs can resume their work, which affected by migration of objects due to contention on within or across node interconnect. We also can see that excessive decomposition makes the performance worse by increasing application time while reducing load balancing overhead.

Our approach is very well suited to handle this limitation of periodic load balancing presented above. We use implicit tasks



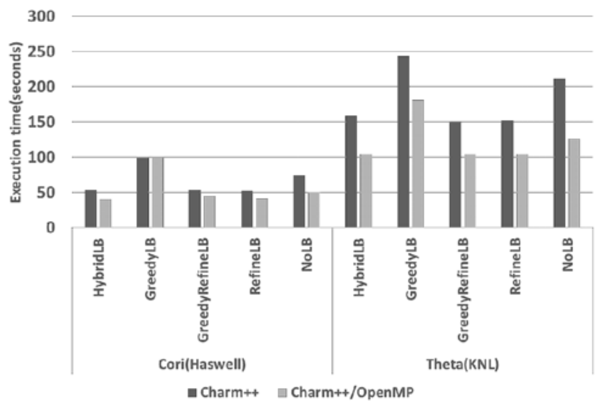
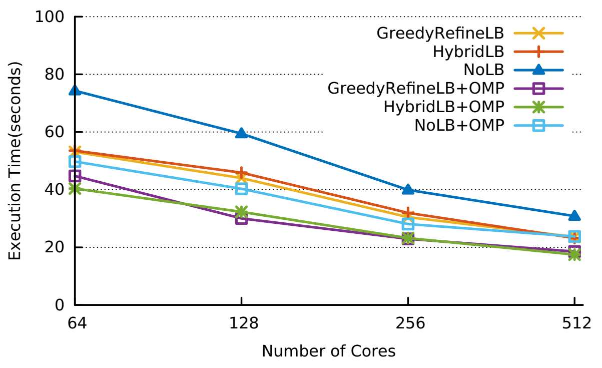
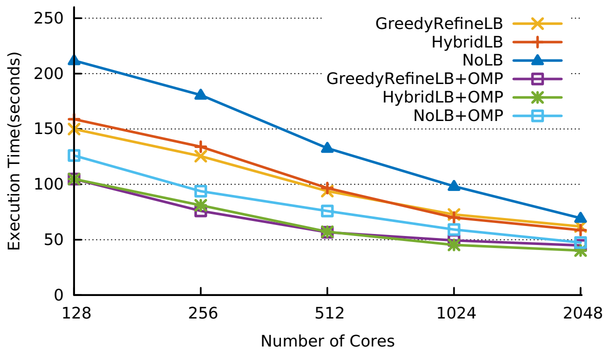


Fig. 7: Improvement of load imbalance and performance of Lassen on a single node of Cori and Theta.



(a) Strong scaling on Cori(Haswell nodes).



(b) Strong scaling on Theta(KNL nodes).

Fig. 8: Strong scaling results of Lassen on Cori and Theta.

generated via our OpenMP integration to resolve existing load imbalance without a significant overhead. Figures 7a and 7b show how much the integrated OpenMP resolves load imbalance and improves the performance of Lassen on a single node of Cori and Theta with best configuration(4 chares/PE, 10 iterations) from Figure 5. We see that excess load is spread to other PEs, which reduces the load imbalance factor. This improvement in load imbalance results in improvement of performance with all load balancing strategies we choose. In addition, Theta shows bigger improvements in load imbalance and performance because Xeon Phi has more cores. Load imbalance can therefore be higher and can get resolved better on Theta with the help of many idle PEs.

We run Lassen to show the benefit of our work on Cori and Theta with and without best performing load balancing strategies such as GreedyRefineLB and HybridLB. Even though HybridLB is worse than RefineLB on a single node, it works well on multi-node runs because of its distributed design. Figure 8 shows how much Lassen is improved. The chosen periodic load balancing schemes can distribute load imbalance across nodes quite well. However, as we noted in the motivation of this work, persistent load balancing alone cannot redistribute all of the existing load imbalance because of its more significant overhead. Figure 8 shows how the OpenMP integration can help distribute this load imbalance within each node. Even only with the OpenMP integration, the load imbalance in Lassen is quite well redistributed and the performance is improved around 29.6% on Cori with 512 cores and 46.5% on Theta with 2K cores. Users can easily resolve load imbalance in their application by adding simple flags of OpenMP while they can redistribute load imbalance across nodes by using persistent load balancing manually. When integrated OpenMP is used with the best performing periodic LB, HybridLB, the performance is improved 32.5% on Cori with 512 cores and 45.9% on Theta with 2K cores.

B. *Kripke*

Kripke [37] is an LLNL proxy application for parallel deterministic transport codes. It is written using MPI and, optionally, OpenMP for parallelism. Kripke implements the key computation and communication aspects of a production transport simulation application. Such codes are used to deterministically solve for the flux of neutral particles within a volume of interest. Kripke implements parallel sweeps through a 3D domain. The domain is decomposed into spatial zones, and subdomains are distributed to MPI ranks.

Parallel sweeps are vital communication kernels for the performance of deterministic transport codes. A sweep is a sequential traversal through a domain. Because of the sequential dependencies through the domain, and because the domain is decomposed spatially, scaling sweeps efficiently is challenging. Consequently, Kripke pipelines successive sweeps over the different energy groups and directions in the problem

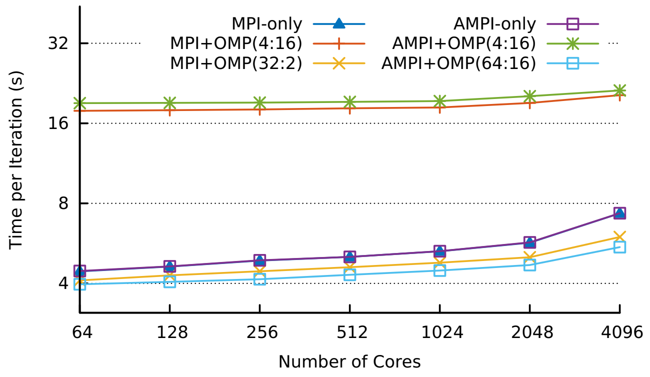


Fig. 9: Weak scaling Kripke with 4096 spatial zones per core on Theta, the time per iteration is shown for MPI and AMPI with and without OpenMP. Numbers in parentheses indicate how many ranks were used per node and the number of OpenMP threads per rank.

to attain higher efficiency. In addition to the sweep, a reduction is performed every iteration to test the global particle count for convergence.

Adaptive MPI (AMPI [5]) is an implementation of the MPI standard on top of Charm++. It provides the high-level features of Charm++, such as over-decomposition, dynamic load balancing, and automatic fault tolerance, to pre-existing MPI applications. It does so by implementing MPI ranks as lightweight, migratable user-level threads, which are encapsulated in chares. The runtime schedules and load balances AMPI ranks the same way as chares in Charm++ programs.

Our OpenMP runtime can be used with AMPI+OpenMP programs the same way it is with Charm++ applications. This allows users to run an AMPI code on a node with N PEs using N or more AMPI ranks per node with each rank using up to N OpenMP threads, without actually oversubscribing the physical resources on the system.

All of the tests below were performed on Theta, using 64 cores per node. We use the default input parameters for Kripke version 1.1. No changes are necessary to the source code of Kripke to run it on AMPI and our implementation of OpenMP. We show weak scaling in the number of zones, with the number of groups and directions held constant.

Figure 9 shows the time per iteration of Kripke using MPI, MPI+OpenMP, AMPI, and AMPI+OpenMP with two different configurations. The parenthetical in MPI+OMP (4:16) and others identifies how many ranks were launched per node, and how many threads may execute any OpenMP parallel-for loop at a time. Thus, MPI+OMP (4:16) indicates the use of 4 ranks per node with 16 OpenMP threads per rank. In addition to MPI-only, AMPI-only, and both with four processes and 16-way threading per node, we show the best performing combination of rank and thread counts for each.

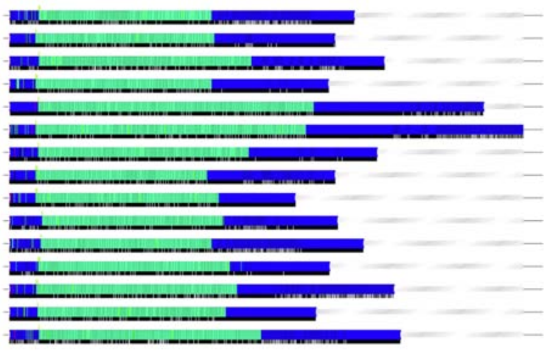
Kripke's parallel sweeps benefit from the finer-grained pipeline parallelism that decomposing into more MPI ranks offers. On the other hand, the computational kernels benefit from OpenMP threading. Since sweep dependencies translate to idle times within a node while each wavefront passes through the domain, within-node parallelism can be also be used to balance the load across the idle threads at a given time. Persistence-based load balancing does not help Kripke's performance, since across iterations the load is balanced. The combination of 64 ranks and up to 16-way threading per rank performs 11% better than the next best combination. Essentially, the AMPI+OpenMP (64:16) case gives the runtime the most freedom to schedule work across all available cores on a node while still decomposing the sweep pipeline into small pipeline stages. These results show the benefits of our unified runtime approach for applications which have transient load imbalances within iterations but little to no load imbalance that evolves and persists across iterations. They also exemplify how our approach can benefit MPI applications.

C. ChaNGa

ChaNGa is an N-body cosmology simulation application implemented in Charm++. ChaNGa has been used in cosmology research to model the impact of a dwarf galaxy on the Milky Way [38], study the role of Warm Dark Matter in dwarf galaxy formation [39] and model the intracluster gas properties in merging galaxy clusters. ChaNGa uses adaptive time scales for force evaluation at multiple scales. A wide variation in mass densities results in particles having dynamical times that vary by a large factor. The irregular distribution of particles in the simulation space as well as having multiple scales creates severe load imbalance. Performing frequent load balancing by object reassignment has unacceptable overhead due to strategy time and data movement. In addition, for clustered datasets, it is often the case at the trailing end of the gravity calculation that some of the PEs are idle while others are busy. For our experiments we use a benchmark dataset *dwf1.2048* which is a highly clustered 5 million particles representing a high resolution dwarf galaxy. In this multi-stepping run of *dwf1.2048* dataset, 16 substeps constitute a big step.

Figure 10a shows the Projections [40] time-line view of this simulation on a single node of Theta with 128 hyperthreads. We pick only a subset of cores within an SMP node for one of the substeps to showcase the load imbalance problem. The colored bars indicate that the PE is busy with computation work and the white shows idle time. X axis represents load and Y axis represents #PE. We can see that clearly there is severe load imbalance. We use the task parallelization in conjunction with the node aware load balancer to handle this load imbalance. With the intra-node task parallelization, we are able to handle the load imbalance and improve the performance of this substep significantly. In figure 10b we can see the impact of this in the reduction of load imbalance, idle time and step time before the barrier.

At the point where the application creates fine-grained tasks, it queries the adaptive runtime system to find out whether it is beneficial to create tasks. The runtime system monitors the state of the PEs on a node and when there are sufficient idle PEs, it considers it as beneficial to create tasks. This prevents incurring unnecessary overhead of task creation when there is


(a) Without intra-node load balancing.

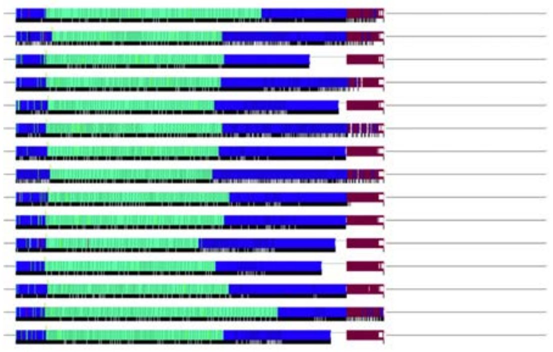

(b) With intra-node task parallelism.

Fig. 10: Time line profile of ChaNGa for all the PEs (rows) on a SMP process for the single node run of Theta with 128 hyperthreads.

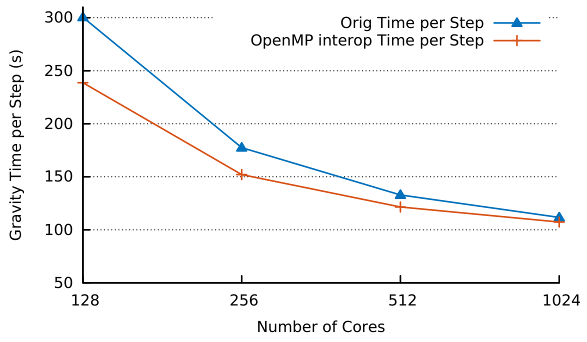


Fig. 11: ChaNGa strong scaling performance on Theta.

no potential benefit because other PEs are already busy. The chare object uses OpenMP to create tasks out of the unfinished buckets which are distributed among other idle cores.

Figure 11 compares the strong scaling performance of the original version of ChaNGa with the OpenMP interoperation version. The input data *dwf1.2048* has strong scale limit so you can see the performance converges as the number of cores increases. On a single node, the integrated OpenMP shows around 25.7% of improvement, 16.7% on 2 nodes, 9.4% on 4 nodes and 4% on 8 nodes compared to Charm++ only.

VIII. Conclusions

The recent trend of rapid increase in the number of cores per chip has resulted in vast amounts of on-node parallelism. Not only the number of cores per node is increasing substantially but also the cores are becoming heterogeneous. The high variability in the performance of the hardware components introduces imbalance. Applications are also becoming more complex resulting in dynamic load imbalance. Load imbalance can result in loss of performance and decrease in system utilization. We address the challenge of balancing load across cores while maintaining locality and low overhead.

In this paper, we proposed a new integrated runtime system that combines the Charm++ distributed programming model with concurrent tasks to handle load imbalance. It utilizes a relatively infrequent periodic assignment of work to cores based on load measurement, in combination with user created tasks to handle both the persistent and transient load imbalance. We integrate OpenMP with Charm++ so as to enable objects to create potential tasks via OpenMP's parallel loop construct. Our contribution is not specific to Charm++. It is also available to MPI applications through AMPI.

We show the benefit of using this integrated runtime system on three different applications. We show the benefit of OpenMP integration on a Charm++ mini app, Lassen with 4 existing load balancing strategies and without load balancing strategy in strong scaling experiments with up to 512 cores on Cori and 2,048 cores on Theta. We also show the benefit on an MPI proxy application, Kripke, in weak-scaling experiments on up to 2,048 cores using Adaptive MPI. We show improvements of 25.7% on ChaNGa with multistepping runs of *dwf1.2048* on a single node and the improvement becomes smaller because of its strong scaling limit.

The task generation scheme we used currently admits relatively flat set of tasks generated by parallel loops. A possible future extension is to admit tasks with dependences, similar to OmpSs [17], PaRSEC [41] etc. These will also create opportunities for runtime scheduling based on the knowledge of dependencies and cache or scratchpad availability of data.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-746241). This material is based in part upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0002374.

References

[1] E. Shmueli, G. Almasi, J. Brunheroto, J. Castanos, G. Dozsa, S. Kumar, and D. Lieber, "Evaluating the effect of replacing CNK with Linux on the compute-nodes of Blue Gene/L," in *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ser. ICS '08. New York, NY, USA: ACM, 2008, pp. 165–174.

[2] S. M. Kelly and R. Brightwell, "Software architecture of the light weight kernel, Catamount," in *Proceedings of the 2005 Cray User Group Annual Technical Conference*. Citeseer, 2005, pp. 16–19.

[3] F. Petrini, D. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q," in *ACM/IEEE SC2003*, Phoenix, Arizona, Nov. 10–16, 2003.

[4] B. Acun, P. Miller, and L. V. Kale, "Variation among processors under turbo boost in hpc systems," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: ACM, 2016, pp. 6:1–6:12.

[5] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel programming with migratable objects: Charm++ in practice," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 647–658.

[6] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[7] V. Kale, A. Bhatele, and W. D. Gropp, "Weighted locality sensitive scheduling for mitigating noise on multicore clusters," in *18th annual IEEE International Conference on High Performance Computing (HiPC 2011)*, December 2011.

[8] E. Demaine, "A threads-only MPI implementation for the development of parallel programs," in *In: Proceedings of the 11th International Symposium on High Performance Computing Systems*, 1997, pp. 153–163.

[9] K. Shen, H. Tang, and T. Yang, "Adaptive two-level thread management for fast MPI execution on shared memory machines," in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, ser. SC '99. New York, NY, USA: ACM, 1999.

[10] H. Tang, K. Shen, and T. Yang, "Program transformation and runtime support for threaded MPI execution on shared-memory machines," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 4, pp. 673–700, Jul. 2000.

[11] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. W. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, "Leveraging MPI's one-sided communication interface for shared-memory programming," in *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 132–141.

[12] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, "MPI+MPI: a new hybrid approach to parallel programming with MPI plus shared memory," *Computing*, vol. 95, no. 12, pp. 1121–1136, 2013.

[13] A. Friedley, G. Bronevetsky, T. Hoefler, and A. Lumsdaine, "Hybrid MPI: efficient message passing for multi-core systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 18.

[14] L. Smith and M. Bull, "Development of mixed mode MPI / OpenMP applications," *Scientific Programming*, vol. 9, no. 2,3, pp. 83–98, Aug. 2001.

[15] E. Ayguade, M. Gonzalez, X. Martorell, and G. Jost, "Employing nested OpenMP for the parallelization of multi-zone computational fluid dynamics applications," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004, p. 6.

[16] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," in *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, ser. PDP '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 427–436.

[17] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, "Productive cluster programming with ompss," in *Euro-Par 2011 Parallel Processing*. Springer, 2011, pp. 555–566.

[18] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur, "Hybrid parallel programming with MPI and unified parallel C," in *Proceedings of the 7th ACM international conference on Computing frontiers*. ACM, 2010, pp. 177–186.

[19] J. Corbalan, A. Duran, and J. Labarta, "Dynamic load balancing of MPI+OpenMP applications," in *Parallel Processing, 2004. ICPP 2004. International Conference on*. IEEE, 2004, pp. 195–202.

[20] V. Kale, A. Randles, and W. D. Gropp, "Locality-optimized mixed static/dynamic scheduling for improving load balancing on SMPs," in *Proceedings of the 21st European MPI Users' Group Meeting*. ACM, 2014, p. 115.

[21] V. Kale, S. Donfack, L. Grigori, and W. D. Gropp, "Lightweight scheduling for balancing the tradeoff between load balance and locality," 2014, poster presented at SC'14.

[22] S. Donfack, L. Grigori, W. D. Gropp, and V. Kale, "Hybrid static/dynamic scheduling for already optimized dense matrix factorization," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 496–507.

[23] M. Prache, P. Carribault, and H. Jourdren, "MPC-MPI: An MPI implementation reducing the overall memory consumption," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 16th European PVM/MPI Users Group Meeting (EuroPVM/MPI 2009)*, ser. Lecture Notes in Computer Science, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2009, vol. 5759, pp. 94–103.

[24] M. Pérache, H. Jourdren, and R. Namyst, "MPC: A unified parallel runtime for clusters of NUMA machines," in *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par 08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 7888.

[25] P. Carribault, M. Prache, and H. Jourdren, "Enabling low-overhead hybrid MPI/OpenMP parallelism with MPC," in *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, Proceedings of the 6th International Workshop on OpenMP (IWOMP 2010)*, ser. Lecture Notes in Computer Science, M. Sato, T. Hanawa, M. Mller, B. M. Chapman, and B. de Supinski, Eds. Springer Berlin Heidelberg, 2010, vol. 6132, pp. 1–14.

[26] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," in *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, Santa Barbara, California, Jul. 1995, pp. 207–216, mIT.

[27] C. Pheatt, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[28] OpenMP ARB, "OpenMP application program interface version 3.0," in *The OpenMP Forum, Tech. Rep*, 2008.

[29] R. Barik, Z. Budimlic, V. Cave, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan *et al.*, "The Habanero multicore software research project," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. ACM, 2009, pp. 735–736.

[30] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "Slaw: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems," in *ACM Sigplan Notices*, vol. 45, no. 5. ACM, 2010, pp. 341–342.

[31] C. Mei, "Message-driven parallel language runtime design and optimizations for multicore-based massively parallel machines," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2012.

[32] L. V. Kalé, "The virtualization model of parallel programming : Runtime optimizations and the state of art," in *LACSI 2002*, Albuquerque, October 2002.

[33] H. Menon, S. Bak, S. White, M. Diener, and L. Kale, "Handling transient and persistent imbalance together in distributed and shared memory," in *PPL Technical Reports 2016*, no. 16-19, December 2016.

[34] S. Bak, H. Menon, S. White, M. Diener, and L. Kale, "Integrating openmp into the charm++ programming model," in *Proceedings of the Third International Workshop on Extreme Scale Programming Models and Middleware*, ser. ESPM2'17. New York, NY, USA: ACM, 2017, pp. 4:1–4:7.

[35] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2005, pp. 21–28.

[36] O. Pearce, T. Gamblin, B. R. de Supinski, M. Schulz, and N. M. Amato, "Quantifying the effectiveness of load balance algorithms," in *26th ACM international conference on Supercomputing*, ser. ICS '12, 2012, pp. 185–194.

[37] A. J. Kunen, T. S. Bailey, and P. N. Brown, "KRIPKE - a massively parallel transport mini-app," Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep., 2015.

[38] C. W. Purcell, J. S. Bullock, E. J. Tollerud, M. Rocha, and S. Chakrabarti, "The Sagittarius impact as an architect of spirality and outer rings in the Milky Way," *Nature*, vol. 477, pp. 301–303, Sep. 2011.

[39] J.-h. Kim, T. Abel, O. Agertz, G. L. Bryan, D. Ceverino, C. Christensen, C. Conroy, A. Dekel, N. Y. Gnedin, N. J. Goldbaum *et al.*, "The agora high-resolution galaxy simulations comparison project," *The Astrophysical Journal Supplement Series*, vol. 210, no. 1, p. 14, 2013.

[40] L. Kalé and A. Sinha, "Projections : A scalable performance tool," in *Parallel Systems Fair, International Parallel Processing Sympos ium*, Apr. 1993, pp. 108–114.

[41] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, "Parsec: Exploiting heterogeneity to enhance scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.