Handling Transient and Persistent Imbalance Together in Distributed and Shared Memory

***Abstract*—The recent trend of rapid increase in the number of cores per chip has resulted in vast amount 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 introduce imbalance due to heterogeneity. The 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 handling both transient and persistent load imbalance while maintaining locality and incurring low overhead. In this paper, we propose a new integrated runtime system that combines the Charm++ distributed programming model with concurrent tasks to handle the load imbalance problem. It utilizes an infrequent 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++ so as to enable creation of potential tasks via OpenMP's parallel loop construct. This is not specific to Charm++ and is also available to MPI applications as well through Adaptive MPI implementation. We show the benefit of using this integrated runtime system on three different applications. We show improvements of 2X on ChaNGa on 128K cores and more than 3X on NAMD at 2K cores. We also show the benefit on an MPI application, Kripke, using Adaptive MPI.**

I. Introduction

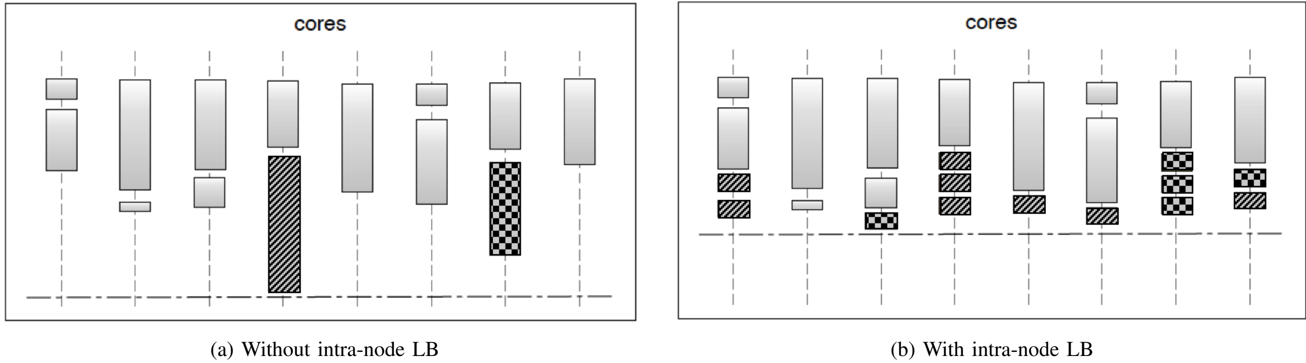
Several trends in high-performance computing are converging to drive applications and systems software to rely on multithreading in each node's shared memory, rather than running an independent process on each CPU core. Increasing per-chip concurrency creates pressure on system memory, system software, and application design. The general abandonment of specialized OS kernels [1], [2] in favor of general-purpose Linux has rolled back past efforts to reduce system noise [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 distributed programming models with both standard OpenMP and new parallel loop and concurrent task constructs that addresses many of these challenging trends with a low-overhead and locality-conscious design.

The number of cores and 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 work 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 are 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.

Even with very balanced work assignment across nodes and individual cores, execution may not proceed at a perfectly uniform 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 [5]. Dynamic work redistribution can greatly help in mitigating these effects [6].

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 both standard OpenMP and new parallel loop and concurrent task constructs. 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 and low and proportionate scheduling overhead. We demonstrate this design's effectiveness through the improved performance and scalability of several applications run on large supercomputers.



(a) Without intra-node LB

(b) With intra-node LB

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

The contributions of the paper are:

- An approach that combines infrequent global load balancing with shared-memory task parallelism to handle transient and persistent load imbalance.
- 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++ to enable fine-grained parallelism.
- Improved performance by using the integrated runtime system on three different applications. We show improvements of 2X on ChaNGa on 128K cores and more than 3X on NAMD at 2048 cores. We also show the benefit on an MPI application, Kripke, using Adaptive MPI.

II. Related Work

Per-chip core and thread counts are steadily increasing in HPC systems. The trend toward increasing core/thread counts will accelerate 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. The ubiquitous MPI has correspondingly evolved in usage and implementation to work well in this setting [7]–[12], 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. [13]–[15]). This hybrid model has been increasingly used with other shared memory programming models to handle within node parallelism [13], [16], [17]. Similar work has been done with Charm++ as the distributed substrate, combined with both OpenMP[1] and the bespoke 'CkLoop' loop-multithreading mechanism [18], both of which we extend in the present work.

The MPI+X model on its own has been shown to 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. A recent set of papers by V. Kale, Gropp, et al. have explored the hybrid model in more detail. They mix static and dynamic scheduling of work among cores on a node to improve the tradeoffs among overhead, locality, and load imbalance [20]–[22]. They also show that these techniques can be used to reduce the impact of system noise [6]. Our work carries these ideas further, by adaptively tuning the level of dynamic scheduling to match its potential utility, thus pushing overhead lower.

Projects to more tightly integrate various shared and distributed memory models have also arisen, with aims to improve scheduling and locality further. OmpSs 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.

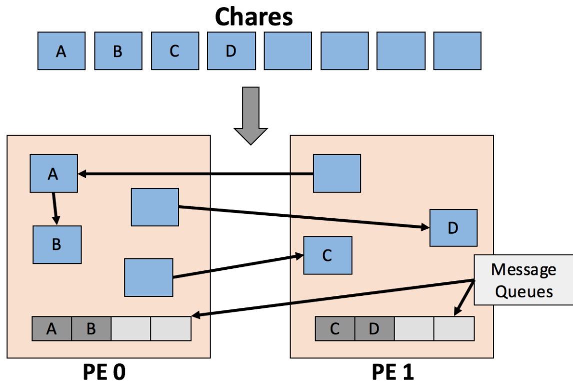


Fig. 2: Charm++ Parallel Programming System

III. Charm++ programming model for shared memory

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 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 communicating via messages.

The SMP mode of Charm++ takes advantage of multi-core shared memory processors [18]. In this mode, a Charm++ OS process is called an SMP node which launches multiple threads and 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. We use core, hardware thread and PE interchangeably. These threads (PEs) have CPU affinity, i.e. each PE is bound to a specific core and the operating system is not allowed to migrate the thread 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 the PEs via pointers. Utilizing the shared memory multi-core processor 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

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 nullifies 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. 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 two methods for users to create potential tasks. The first method is a task abstraction that we have added to Charm++. The second one, which builds on it, is an integration of OpenMP with Charm++, such that each object can create potential tasks via OpenMP parallel loop constructs. Both of these are capable of creating potential tasks that can be used for dynamically utilizing 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 the periodic load balancing in Section V. Then we describe the task model in Charm++ in Section VI. Following this we describe our OpenMP integration with Charm++ in Section VII. Finally, we showcase the application performance improvements achieved by using the new integrated runtime system in Section VIII.

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* [31] holds which says that the communication pattern and computation load of the recent

past is a good indicator of near future. We use this to predict the load of future iterations; The predictions are used by the load balancing strategies to make the global decisions. We work with Charm++ because of the support for dynamic load balancing. As mentioned in the earlier section, in Charm++, the data and its computation is encapsulated into a chare object which resides on a specific PE. A PE here refers to a processing element such as a core or a hardware thread. 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 two-level load balancing strategy for one of the applications, ChaNGa: the load is first balanced across nodes and then balanced within each node, both by assigning chares to PEs. This ensures that the load is distributed evenly among the nodes; it is also distributed evenly among cores of the node to the extent that the load predictions hold. The other load balancing strategy used in this paper is a hierarchical load balancer. 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. Handling Residual and Transient Load Imbalance with Charm++ Task Model

A. Task API

We support two methods of task creation. One is using an API in Charm++ to support loop parallelization. The second one is creation of tasks via OpenMP's parallel loop construct described in Section VII. The following API is provided to the programmer to expose loop parallelism.

```
ParallelFor(funcptr, int argc, void* argv,
            int start, int end, int step,
            int redOp, void *redBuf,
            callback*, int sync)
```

The *funcptr* is the pointer to the function that executes the chunk of work on any core within the node. We support a limited number of reduction operations. If *sync* is set then it does not return control until all the chunks are done executing. If *sync* is not set, then the control returns as soon as all the tasks have been picked by any of the cores. If a *callback* is set, then it invoked once all the chunks of work are completed.

B. Task Generation and Scheduling

A straightforward way to schedule *parallel-for* tasks is to statically assign equal chunks of work to all the cores within the node as done by OpenMP's `static` schedule. This is not suitable for our case where worker threads may be busy with their own computation. If other cores are busy with their computation work, then they won't pick up the statically assigned task to execute. This will result in the delay in completion of the *parallel-for* loop and wastage of CPU cycles at the caller. Alternatively, one could create all the chunks and push them into a common task queue from which other threads will pick work. This could have high overhead of task creation and contention at the shared task queue. We use a separate task queue for each PE which is described in detail in Section VI-C.

Algorithm 1 Recursive Splitting

Input:
low - Lower Index of the Task Array
$high$ - Higher Index of the Task Array
mid - Middle Index of the Task Array
$taskDesc$ - Task Descriptor
$chunkSize$ - Chunk Size

```
 1: function RECURSIVESPLIT(low, high, taskDesc)
 2:     size = high − low
 3:     if (size < chunkSize) then
 4:         executeTask(low, high, taskDesc)
 5:         return 0
 6:     else
 7:         Task tPushed = new task(mid, high, taskDesc)
 8:         Push (tPushed)
 9:         RECURSIVESPLIT(low, mid, taskDesc)
10:         Task tPopped = Pop()
11:         if (tPopped = NULL) then          ▷ If Pushed task is stolen
12:             return 0
13:         else                              ▷ If Pushed task is not stolen
14:             RECURSIVESPLIT(mid, high, taskDesc)
15:         end if
16:     end if
17: end function
```

We explore other task generation and scheduling strategies many of which involve work-stealing such as done in Cilk [26].

1) Recursive ParallelFor Task Generation: Algorithm 1 describes the algorithm for recursive ParallelFor. In this mode, one task descriptor is created with all the information about the task. Typically the task descriptor contains the object pointer, function pointer, total number of chunks and an atomic variable to keep track of the number of finished chunks. In recursive ParallelFor task generation, the loop iterations are split into two halves (similar to the Cilk recursive spawn). A task message is created for one of the halves. This task message contain the iteration range and a pointer to the common task descriptor. The worker pushes the task message into the task queue and calls the function recursively for the other half. If the iteration range is within the chunksize, then the task is executed. The thief steals the task from the head of the queue. This ensures that a large fraction of the work is stolen. The thief will then generate more tasks which are added to its task queue and starts working on chunks.

2) Broadcast Task Message: We have a single task descriptor with information about the task. A message containing

pointer to the task descriptor is sent to all the PEs within the node via a broadcast tree. Whenever the scheduler on a PE picks up the message it repeatedly and atomically increments a variable to get the next chunk to work on and executes that chunk of work, until there are no chunks left to schedule.

3) Only When Idle: A PE incur unnecessary overhead due to task creation and queue contention when there are no idle PEs who can steal and execute some of their tasks. We use an atomic counter to keep track of the number of idle PEs within the node. Any PE trying to generate fine-grained tasks can use this information to decide number of tasks to generate thereby adaptively controlling the number of tasks generated depending on the state of the system.

4) History: We utilize the *principle of persistence* to further reduce the overhead of task creation. Each PE keeps a history of fraction of tasks that was locally executed. This information is used to decide the number tasks to be generated and pushed to the task queue to enable work sharing.

C. Task Queue

To support tasks, we created a task queue on each PE, which is distinct from the normal message queue. The messages in the message queue are meant for that specific PE whereas the tasks in the task queue can be distributed across different cores on a node. The scheduler on the PE polls the local task queue and the message queue for messages. We chose not to have a centralized task queue at the node level because then we lose locality information and there could be potential contention for the centralized queue. We have a separate task queue on each PE, which is a single producer multiple consumer queue for the fine-grained tasks. Whenever a PE becomes idle, it randomly chooses a PE and picks tasks from that PE's task queue. This is similar to Cilk's workstealing [26], except that our scheduler also polls other queues, including a PE-specific message queue for messages to chares assigned to that PE by the periodic load balancer.

The task queue is implemented using the Chase-Lev [32] non-blocking algorithm. The task queue is a double-ended queue. A push(t) call enqueues a task at the tail of the queue. A pop() call dequeues a task from the tail of the queue. A steal() call dequeues from the head of the queue. The queue is a cyclic array of task pointers with non-wrapping head and tail indices. A worker does a push(t) by adding the task at the tail of the queue and increments T, the tail pointer. A worker does a pop() by decrementing T. If it detects that there could be a conflict, then it uses compare and swap (CAS) to handle the conflict. A thief reads H and T and uses CAS to atomically increment H and obtains the task.

The task descriptor contains details about the task such as the object pointer, function pointer, parameters and an atomic variable. The message enqueued into the task queue contains range parameters and a pointer to the common task descriptor. To avoid the overhead of creation of messages and task descriptors, we keep a pool of task messages and descriptors which are reused.

VII. OpenMP interoperation with Charm++

In this section, we discuss the OpenMP thread model and our implementation and optimization of its runtime features for Charm++.

A. OpenMP thread model

OpenMP is the de-facto standard for task-level parallel programming on shared-memory systems. It has been widely adopted and implemented on existing platforms. OpenMP was originally based on the fork-join model of parallelism [33]. All OpenMP programs start with a single thread, called the initial thread. This thread runs sequentially and serves as a host. The initial thread can offload its task into other threads on the same node. When this initial thread encounters a *parallel* pragma, it constructs a team where all threads on the same node keep information of OpenMP tasks assigned to them. After this *parallel* region ends, the initial thread continues execution. While this initial thread is running in a sequential region, all other threads wait in a thread dock for the next task. While initially based on a synchronous model, the latest versions of the OpenMP standard have added more asynchronous features.

B. Implementation of OpenMP for Charm++

Common OpenMP runtime system would spawn their own threads independent of Charm++ worker threads. Without proper coordination between the two runtime systems the OpenMP and the Charm++ threads will contend for hardware resources and lead to oversubscription of cores. To enable OpenMP to efficiently work with Charm++, we modified an OpenMP library to use Charm++ threads, so that the two runtimes can share resources.

We used GNU OpenMP 4.0, which is forked from GCC 4.9.3 for better portability and support across many platforms. First, we modified the OpenMP runtime to use Charm++ threads to execute its tasks. Instead of spawning new threads for the execution of OpenMP tasks, our OpenMP runtime puts the task descriptor in Charm++ message. These messages are pushed into a thread-local task queue that can be accessed by other threads on the same node. The idle threads steal the tasks from the task queue. Because OpenMP is predominantly a synchronous programming model, all OpenMP programs have an implicit synchronization point in termination. Without removing these implicit synchronization points, the OpenMP tasks would make all Charm++ threads wait at a number of barriers. As all threads in Charm++ are both worker as well as master threads, removing these barriers is necessary because otherwise it can lead to a hang. To solve this issue, we eliminate all barriers in OpenMP and replace them with atomic counters for each OpenMP task collection. When a chare generates OpenMP tasks, it records the number of tasks in its own team structure. Then, when other chares attempt to steal tasks from a busy thread, they decrement the appropriate counter to notify the master thread that its task is going to be executed. All OpenMP tasks pushed into the task queue can now be considered normal Charm++ messages, which can be executed and migrated within a node. The Figure 3

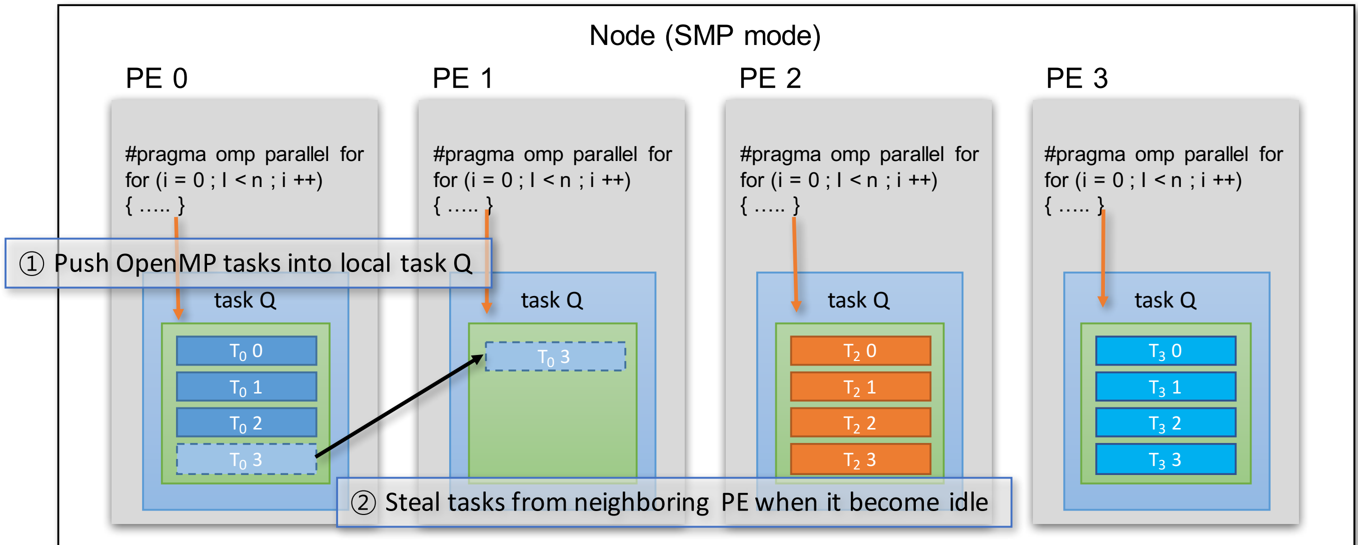


Fig. 3: Implementation of OpenMP for Charm++ using the task API

shows how the OpenMP interoperates with Charm++ when Charm++ runs on a node with 4 PEs and use *static* scheduling to split each chare's task into OpenMP tasks. For the purpose of simplicity, we show how the static schedule of OpenMP works in this integrated runtime system. First, each chare splits its task into as many OpenMP tasks as there are PEs on a node. The OpenMP runtime puts each OpenMP task in a Charm++ message and pushes all of the messages into the thread local task queue. An idle thread can potentially steal a task from one of busy threads on the same node, thereby distributing the work.

C. Scheduling schemes of OpenMP for Charm++

1) Basic scheduling schemes for OpenMP: The number of messages created for OpenMP tasks resulted in overheads in message creation and queue contention. We identified various opportunities for performance improvement and implemented them as different scheduling schemes. In the OpenMP standard, there are four kinds of scheduling schemes for OpenMP tasks. The first and default scheduling policy in many implementations is *static* scheduling. *static* scheduling assigns the iterations of a for-loop to cores in blocks of size number of iterations divided by the number of physical threads in a node. This incurs no overhead due to task creation and contention because it is done by the compiler. In the *dynamic* schedule, threads in a team pick and execute next available iterations. Dynamic scheduling incurs some overhead due to task creation, contention of shared resources as well loss of locality. In the *guided* policy, each thread in the team is assigned a chunk of iteration proportional to the number of unassigned iterations divided by the number of threads in a team. Whenever each thread in a team finishes its assigned task, the next assigned chunk is determined in this way. User can specify the minimum size of chunk in the *guided* policy. The *auto* policy is specific to each implementation.

2) Changing the portion of stealable OpenMP tasks: We first consider static scheduling and show how we minimize the overheads of our task scheduler. Although static scheduling avoids the runtime overhead of dynamic and guided policies, static scheduling can still cause significant overhead by the creation of excessive numbers of messages. We implemented some of ideas described in Section VI-B to avoid this overheads. To minimize overheads of accessing the local task queue, we make all threads have a history vector to keep a record of previous ratios of stolen tasks and locally executed tasks. Using the moving average of the previous ratios in the history vector helps each thread decide how many of the generated tasks it should push into its local task queue for work stealing. This can reduce some overheads for each thread to push and pop its own OpenMP messages into the local task queue.

3) Changing the number of OpenMP messages created: We use an atomic counter for the number of idle threads in the Charm++ runtime to prevent each thread from creating more messages than the number of idle threads. This can reduce overheads in creating messages significantly and efficiently. When the OpenMP runtime splits each thread task into OpenMP tasks, it first inspects the idle counter maintained by the runtime system. In addition to this value, the OpenMP runtime also looks at the local history record of previous ratios of work stolen. These ratios represents how many of tasks have been stolen by other threads. Then, when each thread needs to split their task into at least the number of messages proportional to the average of these previous ratios. In our integration of OpenMP for Charm++, we use a bigger value of average ratio in the history vector and the number of idle threads in the atomic counter to decide how many messages to be created. Using only the counter may restrict parallelism at times because each thread may lose the opportunity to receive help from other threads becoming idle while its tasks are being

executed.

VIII. Application Study

We study the performance benefits of our new integrated runtime system that combines the Charm++ distributed memory model with the task model on two production scientific simulation codes, ChaNGa and NAMD, as well as its use in an MPI+OpenMP proxy application, Kripke. We compare the performance of these codes with and without the task model integrated. We show the performance of ChaNGa on Blue Waters and NAMD on Blue Waters and Blue Gene Q. For all the applications, we picked the scheduling strategy that performed the best. For the Charm++ *ParallelFor*, we use the *recursive task generation* scheme and for OpenMP we use the *history* scheme. Both of them were used in conjunction with the *when idle* strategy.

Blue Waters is a Cray XE/XK hybrid machine hosted by NCSA consisting of AMD 6276 Interlagos processors located at the National Center for Supercomputing Applications (NCSA). It has 22,640 Cray XE nodes and 4,228 Cray XK nodes that include NVIDIA GPUs. On the XE nodes there are two AMD Interlagos 6276 processors processors and each processor has 8 Bulldozer cores. Each Bulldozer Core compute unit has 16 integer cores and 8 floating point cores. Our benchmarks are run entirely on the CPU-only XE nodes. Vesta, which is a Blue Gene Q installation located at Argonne National Laboratory (ANL), has 2048 nodes of 1600 MHz PowerPC A2 cores. Each node has 16 PowerPC A2 cores available to applications with 4 hardware threads per core.

A. 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 [34], study the role of Warm Dark Matter in dwarf galaxy formation [35] 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 challenging dataset *cosmo25* which is a highly clustered 2 billion particle dark matter simulation. In this multi-stepping run of *cosmo25* dataset, 16 substeps constitute a big step.

Figure 4a shows the Projections [36] time-line view of this simulation on 128K cores. 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. We can see that clearly there is severe load imbalance. We use

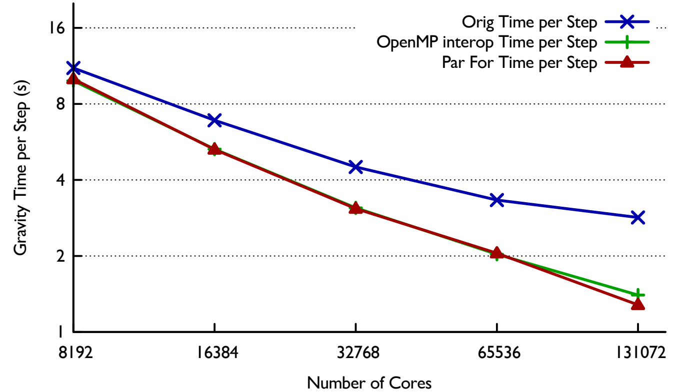


Fig. 5: ChaNGa strong-scaling performance on Blue Waters Cray XE6 system, using Charm++ alone, with integrated OpenMP, and with the *ParallelFor* extension. Both intra-node balancing mechanisms give more than 2X speedup at 128K cores.

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 4b we can see the impact of this in the reduction of load imbalance, idle time and step time before the barrier.

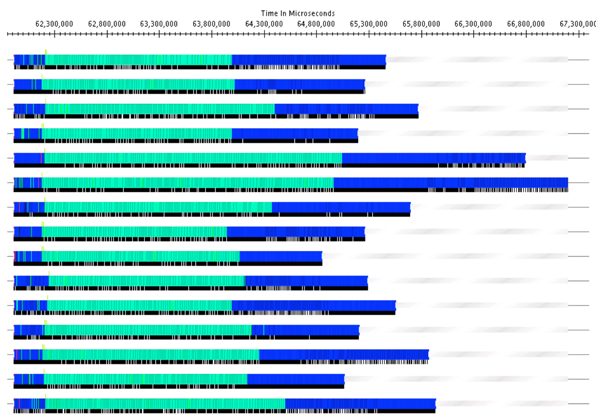
At the point where the application creates fine-grained tasks, it queries the adaptive runtime system to find out whether it is beneficial to create tasks. The runtime system monitors the state of the PEs on a node and when there are sufficient idle PEs, it considers it as beneficial to create tasks. This prevents incurring unnecessary overhead of task creation when there is no potential benefit to it because other PEs are already busy. The chare object uses OpenMP or the Charm++ par for construct to create tasks out of the unfinished buckets which gets distributed among other idle cores.

Figure 5 compares the strong scaling performance of the original version of ChaNGa with the improved one using intra-node fine-grain tasks. At the scale of $131,072$ cores, both *ParallelFor* and OpenMP give more than 2X speedup.

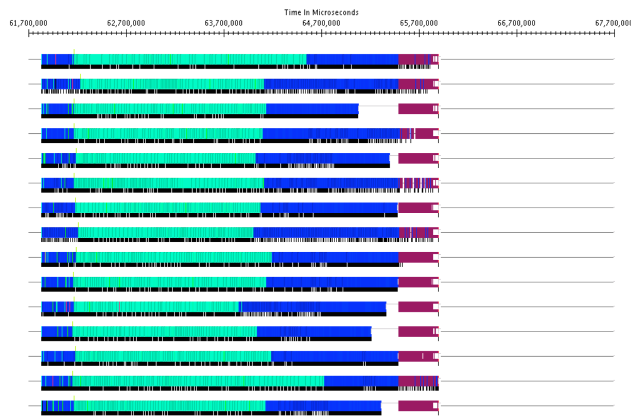
B. NAMD

NAMD [37] is a molecular dynamics application designed for the simulation of large biomolecular systems. Its primary focus is on all-atoms simulation methods using empirical force fields with a femtosecond time step resolution. Typical NAMD simulations include all-atom models of proteins, lipids, and/or nucleic acids as well as explicit solvent (water and ions) and range in size from 10,000 to 10,000,000 atoms. NAMD played an instrumental role in a recent study resolving the atomic level structure of the HIV Capsid. A recipient of the Gordon Bell Award, NAMD is based on Charm++ parallel objects and scales to hundreds of cores for typical simulations and beyond 500,000 cores for the largest simulations.

For the experiments shown here, we use the Colvar module. Colvar stands for *Collective Variables*. Colvars are used to



(a) Without intra-node load balancing



(b) With intra-node task parallelism

Fig. 4: Time line profile of ChaNGa for all the PEs (rows) on a SMP process for the $128K$ cores run. White shows idle time and colored bars indicate busy time. Fine-grained task parallelism achieves better distribution of work among PEs. The total time per step reduces from 5.0 seconds to 4.2 seconds.

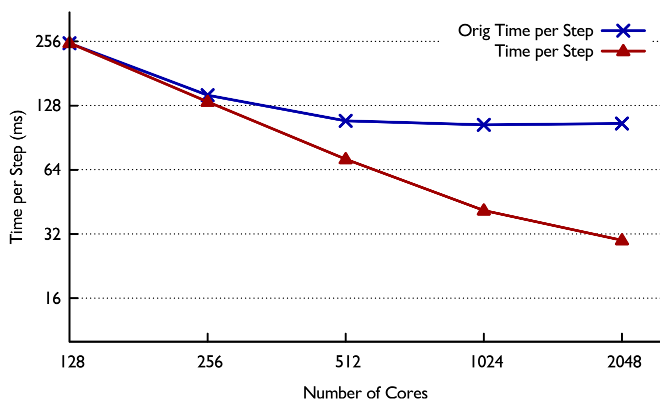


Fig. 6: Strong scaling results comparing the performance of original Charm++ with the new integrated task model for NAMD's colvar benchmark on IBM Bluegene/Q.

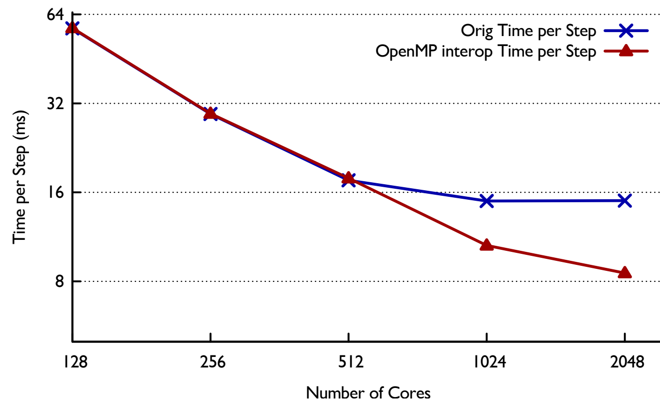


Fig. 7: Strong scaling results comparing the performance of original Charm++ with OpenMP integration for NAMD's colvar banchmark on Blue Waters.

reduce the great number of degrees of freedom present in molecular dynamics simulations into a few parameters which can either be analyzed individually, or manipulated in order to alter the dynamics in a controlled manner. In NAMD, we use the colvar module to perform energy minimization runs and determine the time taken for each step. We use a hierarchical load balancing strategy to infrequently to address the load imbalance problem. For the load imbalance arising within the node, we use our intra-node task parallelization to distribute the computation on idle PEs within a node.

Figure 6 compares the performance of the original version of NAMD running colvar module with the improved version using intra-node fine-grain tasks. At the scale of 2048 cores it gives a speedup of approximately 3.5X. We were unable to run OpenMP integration on Bluegene/Q due to compiler incompatibility.

We also ran NAMD colvar on Blue Waters to compare performance of the original version and the OpenMP interoperation version. Figure 7 shows how much the OpenMP interoperation improves the performance of NAMD colvar. The original version reach its scalability limit at 512 cores, while the OpenMP interoperation version continues to scale up to 2K cores.

C. *Kripke*

Kripke [38] 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 simulate the flux of neutral particles within a volume of interest. Kripke implements parallel sweeps through a 3D domain. The domain is decomposed in 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. Thus, Kripke pipelines successive sweeps over the different energy groups and directions in the problem to attain higher efficiency. In addition to the sweep, a reduction is performed every iteration to test the global particle count for convergence. While Kripke does not actually check for convergence (it instead runs for a fixed number of iterations), the reduction keeps the communication pattern faithful to production transport codes. The reduction acts as a barrier, preventing ranks that own subdomains in the center of the domain from advancing until all ranks have finished all sweeps.

Adaptive MPI (AMPI [39]) is an implementation of the MPI standard written 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 system schedules and load balances AMPI ranks the same way it does chares in Charm++ programs. MPI applications with no mutable global/static variables, such as Kripke, need only be compiled using AMPI's compiler wrappers instead of MPI's to run on AMPI.

Our implementation of the GNU 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 two modes: (a) 1 or a few AMPI ranks per node with OpenMP threads within each rank or (b) 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. Our results show the benefits of this approach for applications such as Kripke which have transient load imbalances within iterations but little to no load imbalance that persists across iterations.

All of the tests below were performed on Blue Waters, using 32 cores per node. We use the default input parameters for Kripke version 1.1, meaning we run with 4096 zones per core in 1 set, 32 groups in 2 sets, and 96 directions in 8 sets. The data is laid out in the default DGZ nesting. Note that no changes are necessary to the source code of Kripke to run it on AMPI and our implementation of OpenMP, and that all of the computational kernels use OpenMP `parallel for` loops. We show weak scaling in the number of zones, with the number of groups and directions held constant.

Figure 8 shows the time per iteration of Kripke using MPI, MPI+OpenMP, AMPI, and AMPI+OpenMP with two different configurations. The parenthetical in MPI+OpenMP (1) and others identifies how many ranks were launched per node. Thus, MPI+OpenMP (1) signifies the use of 1 rank per node with 32 OpenMP threads per rank, and MPI+OpenMP (16) means 16 ranks were launched per node with 2 OpenMP threads per rank. AMPI+OpenMP are similarly presented, but since our OpenMP implementation allows scheduling OpenMP threads along with AMPI ranks without resource contention,

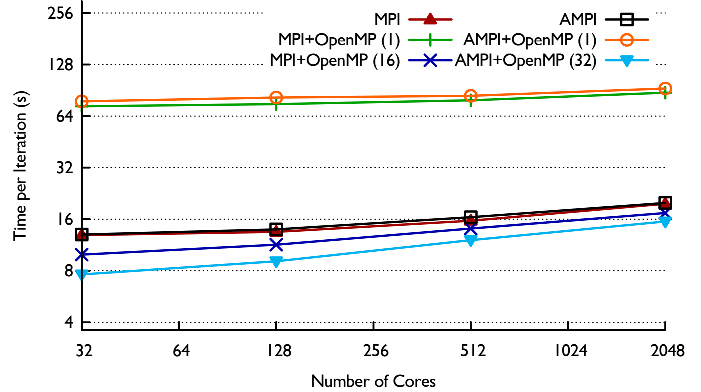


Fig. 8: Weak scaling Kripke with 4096 spatial zones per core on Blue Waters, the time per iteration is shown for MPI and AMPI with and without OpenMP. Numbers in parentheses indicate how many ranks were used per node.

we always specify 32 OpenMP threads per rank. Consequently, AMPI+OpenMP (32) means 32 ranks were launched per node with 32 OpenMP threads per rank. In addition to MPI-only, AMPI-only, and both with one process and 32-way threading, 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. The combination of 32 ranks and up to 32-way threading per rank performs the best. It 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 and ensuring that each thread has its own work to schedule in addition to stealing others' work when idle.

IX. Conclusion

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 due to heterogeneity. 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 integration with Adaptive MPI. We show the benefit of using this integrated runtime system on three different applications. We show improvements of 2X on ChaNGa on 128K cores and more than 3X on NAMD at $2,048$ cores. In these applications, benefit naturally increase with high core counts, when one is nearer to the limit of strong scaling. We also show the benefit on an MPI application, Kripke, in a weak-scaling experiments on up to $2,048$ cores using Adaptive MPI.

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 [16], PaRSEC [40] etc. These will also create opportunities for runtime scheduling based on the knowledge of dependencies and cache or scratchpad availability of data.

X. Acknowledgments

ChaNGa was initially developed under NSF ITR award 0205413. Contributors to the development of the code include Graeme Lufkin, Sayantan Chakravorty, Amit Sharma, Filippo Gioachin, Pritish Jetley, Lukasz Wesolowski and Harshitha Menon. This work was supported by NSF award AST-1312913.

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. [Online]. Available: http://doi.acm.org/10.1145/1375527.1375554

[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. Kalé, "Variation among processors under Turbo Boost in HPC systems," in *International Conference on Supercomputing (ICS)*. ACM, 2016.

[5] 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.

[6] 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.

[7] 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.

[8] 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. [Online]. Available: http://doi.acm.org/10.1145/331532.331581

[9] 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. [Online]. Available: http://doi.acm.org/10.1145/363911.363920

[10] 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. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33518-1_18

[11] 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.

[12] 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.

[13] L. Smith and M. Bull, "Development of mixed mode MPI / OpenMP applications," *Scientific Programming*, vol. 9, no. 2,3, pp. 83–98, Aug. 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=1239928.1239936

[14] 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.

[15] 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. [Online]. Available: http://dx.doi.org/10.1109/PDP.2009.43

[16] 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.

[17] 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.

[18] 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.

[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. Pérache, 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. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03770-2_16

[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. 78–88. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85451-7_9

[25] P. Carribault, M. Pérache, 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. Müller, B. M. Chapman, and B. de Supinski, Eds. Springer

Berlin Heidelberg, 2010, vol. 6132, pp. 1–14. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13217-9_1

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

[32] 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.

[33] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[34] 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.

[35] J.-h. Kim, T. Abel, O. Agertz, G. L. Bryan, D. Ceverino, C. Christensen, C. Conroy, A. Dekel, N. Y. Gnedin, N. J. Goldbaum, J. Guedes, O. Hahn, A. Hobbs, P. F. Hopkins, C. B. Hummels, F. Iannuzzi, D. Keres, A. Klypin, A. V. Kravtsov, M. R. Krumholz, M. Kuhlen, S. N. Leitner, P. Madau, L. Mayer, C. E. Moody, K. Nagamine, M. L. Norman, J. Onorbe, B. W. O'Shea, A. Pillepich, J. R. Primack, T. Quinn, J. I. Read, B. E. Robertson, M. Rocha, D. H. Rudd, S. Shen, B. D. Smith, A. S. Szalay, R. Teyssier, R. Thompson, K. Todoroki, M. J. Turk, J. W. Wadsley, J. H. Wise, A. Zolotov, and f. t. AGORA Collaboration29, "The AGORA High-resolution Galaxy Simulations Comparison Project," *The Astrophysical Journal*, vol. 210, p. 14, Jan. 2014.

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

[37] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten, "Scalable molecular dynamics with NAMD," *Journal of Computational Chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005.

[38] A. J. Kunen, T. S. Bailey, and P. N. Brown, "KRIPKE - a massively parallel transport mini-app," 2015.

[39] 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. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.58

[40] 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.