Parallel Programming with Migratable Objects: Charm++ in Practice

Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida,
Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, Laxmikant Kale

Department of Computer Science, University of Illinois at Urbana-Champaign

***Abstract*—The advent of petascale computing has introduced new challenges (e.g. heterogeneity, system failure) for programming scalable parallel applications. Increased complexity and dynamism in science and engineering applications of today have further exacerbated the situation. Addressing these challenges requires more emphasis on concepts that were previously of secondary importance, including migratability, adaptivity, and runtime system introspection. In this paper, we leverage our experience with these concepts to demonstrate their applicability and efficacy for real world applications. Using the CHARM++ parallel programming framework, we present details on how these concepts can lead to development of applications that scale irrespective of the rough landscape of supercomputing technology. Empirical evaluation presented in this paper spans many mini-applications and real applications executed on modern supercomputers including Blue Gene/Q, Cray XE6, and Stampede.**

I. INTRODUCTION

Parallel computing is at a crossroads. Computational modeling using supercomputers will likely produce breakthrough results in science and engineering in the coming years, utilizing the increasingly powerful parallel machines. Yet, there are significant challenges that need to be overcome. A major category of future challenges involves dealing with dynamic variations. Such variations can occur in the application itself, e.g. when it uses adaptive refinement to increase the resolution in selected subregions of space. Variability can also arise from the machine itself, due to thermal considerations, power constraints, and failures. In the face of these challenges, how can we program the parallel machines of the future to run sophisticated applications productively and efficiently?

It is now widely recognized that an adaptive runtime system (RTS) will be an essential component of the solution to these challenges [1]–[3]. However, we believe that to truly empower runtime systems, the programming models must possess some essential attributes, including over-decomposition, asynchronous message-driven execution, and migratability. The idea of over-decomposition is to divide the computation into many more independent units than the number of processors. Thus, there are multiple work and data units assigned to each processing element (PE) by the RTS. Which of these units should be allowed to execute next on a given PE? Instead of executing them in a pre-programmed sequence, we advocate message-driven execution: control transfers cooperatively from one unit to the other depending on which one is ready to execute. Over-decomposition combined with asynchronous message-driven execution provide multiple benefits that we describe in Section II. Migratability is another essential attribute: the work and data units are not confined to a fixed PE by the program. Instead, the RTS is free to migrate these units to any processing element in the system during execution. This also implies that the programmer writes the code in terms of these logical units, with little reference to the notion of physical execution units such as processors.

As we describe in this paper, these attributes allow us to build a powerful RTS that can dynamically balance load, tolerate component failures, control chip temperatures, constrain power, or minimize energy. It can optimize communication, and shrink or expand the sets of processors used by a job, which is particularly useful in cloud environments.

The CHARM++ parallel programming system [4] has been designed with the above attributes in mind. It realizes the above-mentioned benefits, while staying within the context of the familiar C++ programming language. Specially designated C++ objects, called chares, play the role of work and data units. Chares communicate via asynchronous method invocations, which facilitate message-driven execution. The CHARM++ system has evolved over the past years in the context of science and engineering applications developed using it. The needs of these applications drove the development of specific features in CHARM++, which benefit newer applications.

This paper describes the state of practice of CHARM++, using multiple mini-applications as well as some full-fledged applications. We first state the motivation and utility of the three key attributes: over-decomposition, asynchronous execution, and migratability (§II), and describe how they are realized in CHARM++. Section III presents the runtime capabilities CHARM++ provides based on these attributes. We then describe several mini-applications spanning multiple application domains, show how each mini-app leverages the CHARM++ features, and present performance results attained on large scale supercomputers (§IV). In some cases, we also showcase the performance of the full-fledged CHARM++ application from which the mini-app was derived. We also show how these concepts can be applied to MPI applications using the Adaptive MPI (AMPI) framework provided by CHARM++.

These mini-apps are representative of real applications in that they capture the communication pattern and certain portions of the computation kernel of real applications. Some of the chosen applications and mini-apps have dynamic characteristics, such as adaptive refinement, load imbalance, irregular communication, and inter-modular dependencies, that are increasingly seen in today's science and engineering applications. We show that the CHARM++ programming model design is suitable for a wide range of parallel applications and demonstrate how CHARM++ features were critical in efficiently parallelizing them.

II. Fundamental Design Attributes

We believe that over-decomposition, asynchronous message-driven execution, and migratability are essential to empower a RTS. In this section, we first describe these attributes and their utility, and then present how CHARM++ has been designed based on these attributes.

A. Over-decomposition

Over-decomposition refers to the division of the computation in an application into a large number of work and data units, typically much more than the number of processing elements. Such a decomposition implicitly decouples the programmer's partitioning of the program and data from the low level resources they execute on, e.g. cores. The underlying RTS then maps these work units to PEs as it sees fit (Figure 1).

This model yields many benefits. Decoupling the decomposition of computation from the number of processors allows the programmer to define his program in terms of logical entities that are suitable to his application, independent of the number of processors. This may enhance the programmer's productivity, while also improving application performance by allowing the RTS to intelligently map these units to PEs. Another benefit that originates from such a design is support for the concurrent composition of parallel modules. These parallel modules can be decomposed independently into suitable number of units and be mapped onto the same set of PEs. It provides the benefit of modularity and allows for overlap of communication and computation across modules. Additionally, over-decomposition results in smaller subdomains which may yield better cache utilization.

B. Asynchronous Message-Driven Execution

Message-driven execution is a method of driving control flow of a program in which work units are scheduled on a PE only when a message is received for them, instead of executing work units in a pre-programmed sequence. On the send side, *asynchrony* requires that the source unit does not block for any immediate response from the receiver. Any intended response is sent in the future as a new message and is handled similarly. The RTS actively probes for incoming messages; on receiving a message, the RTS identifies the work unit which the received message targets and schedules it on the PE when possible. In Figure 1, we see that work units A, B, C and D have messages and are waiting to be scheduled by their respective PEs. The remaining work units will not be scheduled until a message is received.

In dynamic scenarios, communication delays can result in idle PEs, which degrades the performance of an application. This is especially true when the communication model forces the sender to wait for the recipient of the message. Over-decomposition with asynchronous message-driven execution can help overcome this common problem. First, the sending work unit does not need to wait for any response and can continue its computation. Second, while a work unit waits for a message, other work units can be scheduled in order to use the PEs efficiently. As a result, this model helps in hiding the latency of communication. It also is useful in decreasing pressure on the network by spreading out communication from different work units over a span of time.

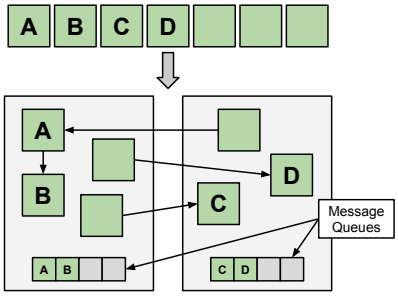


Fig. 1: An example of message-driven over-decomposition.

C. Migratability

Migratability is the ability to move work and data units among the PEs. In combination with over-decomposition, this empowers the RTS to map work and data units to the PEs of its choice for performance. Additionally, the RTS can redistribute the work and data units dynamically during the execution. Dynamic migration enables a number of key benefits that will be detailed in Section III:

— Dynamic load balancing.
— Checkpointing the current state of the application.
— Recovery from hard and soft failures.
— Temperature control and power aware redistribution.
— Malleable job scheduling with shrink and expand.

D. CHARM++

As stated in Section I, the foundation of the CHARM++ parallel programming framework is based on the aforementioned attributes. We now briefly describe how these attributes are embodied in CHARM++ to empower its RTS and enable features presented in Section III.

Over-decomposition: In CHARM++, users define their application in terms of special C++ objects called *chares* and indexed collections of chares called *chare arrays*. For the most part, chares and chare arrays are defined just like standard C++ classes and C++ arrays, allowing the programmer to take advantage of data encapsulation and abstraction. Typically there are many more chares than PEs, allowing the RTS to take advantage of over-decomposition. Because CHARM++ uses an object based paradigm, the programmer can also easily implement different types of work units. A brief example is shown in Figure 2, which shows the decomposition of our LeanMD mini-app. The squares represent Cell objects, and the diamonds represent pairwise Compute objects. A more detailed description of how LeanMD utilizes object based over-decomposition is given in Section IV-B.

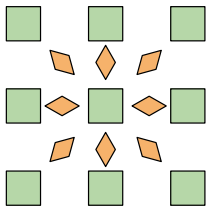


Fig. 2: LeanMD Decomposition

Asynchronous Message-Driven Execution: This attribute is enabled in CHARM++ via two entities — *proxy* and *entry methods*. When the programmer creates chares, the RTS returns a proxy that can be used to refer to any chare in the global space. The programmer also marks certain methods of the C++ objects as *entry* methods in a separate *ci* file. A simple translator provided by CHARM++ reads *ci* files

and generates additional code allowing entry methods to be invoked remotely via the proxy object. To send a message, the programmer simply calls an entry method on a proxy object, and the CHARM++ RTS will eventually invoke that method on the actual chare in the global space.

Migratability: CHARM++ deploys a *packing and unpacking (PUP)* framework to enable migration of chares. In order to make a chare migratable, the programmer defines a member function called *pup*. In this function, the entire object is serialized to or deserialized from a stream of bytes provided by the RTS. As shown in Figure 3, this process has been made convenient by CHARM++ via overloading of the pipe operator for common data types and classes.

```
class A {
  int foo;
  float bar[32];
  void pup(PUP::er&);
};
```

```
void A::pup(PUP::er &p) {
  p | foo;
  PUParray(p,bar,32);
}
```

Fig. 3: Example PUP Function

Scalable Location Management: To support the CHARM++ programming model, the underlying RTS infrastructure, which handles message delivery and location management, needs to be scalable [5]. This is particularly challenging because the RTS needs to manage several collections of chare arrays and find the current location of the over-decomposed objects for message delivery in the face of migrations and faults. For scalable location management, the RTS provides a unique index to every chare created in the system. Depending on the type of chare, the index can vary from being a one-dimensional to six-dimensonal structure or be a user defined name. Every indexed chare is mapped to its *home PE*, which is responsible for managing updated information about it. Several default schemes are provided by the RTS for assigning home PEs to chares, from which the programmer is free to select the most suitable. Programmers can also define their own scheme and guide the RTS to use it.

The RTS uses *location caching* to ensure efficient message delivery to chares. When a new message is sent to a chare, the RTS first checks if it knows the current location of the chare in its location cache. If the location is known, the message is immediately sent to the destination PE. This scheme works well if there is persistence in the interaction pattern of the application, which is true for many scientific codes. In case the current location of a chare is not known, the home PE is queried for its current location. During migration of a chare, the RTS makes sure that the home PE is updated with the current location of the chare. Both the queries and the updates to home PE are fully distributed in order to make location management scalable.

Adaptive MPI: Many of the CHARM++ features discussed can be realized for legacy MPI codes as well. We achieve this through an MPI implementation called Adaptive MPI (AMPI). It is built on top of the CHARM++ framework and uses light-weight user-level threads instead of OS processes. AMPI allows us to virtualize several MPI ranks on a single physical core, which brings the benefits of over-decomposition mentioned previously. In addition, the ranks can be migrated to realize other benefits discussed earlier, such as automatic load balancing and fault tolerance.

III. PROGRAMMING MODEL FEATURES

Building upon the fundamental attributes and their embodiment in Charm++, described in Section II, the RTS can provide many important features required to navigate through dynamic variations in application behavior and machine environment. In this section, we describe these features in detail.

A. Load balancing

Applications are increasingly becoming complex and are relying on irregular structures and adaptive refinement techniques. As a result, the computation load varies dynamically as the execution proceeds. This is exhibited frequently in applications such as molecular dynamics, cosmology simulations, weather simulations, structural dynamics simulations, and adaptive mesh refinement. For such applications, performing load imbalance and deciding its frequency are critical factors that determine the scalability and performance.

As stated earlier, over-decomposition along with migratability empowers the RTS to perform adaptive load balancing. In order to find a good mapping, load balancing strategies deployed by the RTS require an estimate of the load (computation and communication) of the work/data units. In simple cases, this load can be based on a model. However, for many scientific applications, the computation load and the communication pattern change slowly, or suddenly but infrequently. Therefore the recent past is a good predictor of the near future and can be used to predict the load and perform load balance. Since the RTS is orchestrating the scheduling on PEs and the communication between the work/data units, it can instrument the computation load and the communication pattern automatically for each unit in a distributed database. The load balancing strategies can use this information to perform mapping of work/data units to PEs. Additionally, the RTS can observe the application and automate the decision of when to invoke load balancing. This relieves the programmer from making decisions on load balancing related components.

CHARM++ provides a mature load balancing framework with a suite of load balancing strategies comprising of various centralized, distributed and hierarchical schemes for balancing computation load or communication [6]. Depending on the needs of applications, the user can invoke appropriate load balancer. The load balancing framework in CHARM++ instruments each chare as well as records the PE's load. In the *AtSync* mode of load balancing, all the chares pause their execution and call *AtSync*. The load statistics are collected and the user specified load balancing strategy is used to compute the new mapping. Once the load balancing decision is made, the framework handles the migration of the chares to the newly mapped PEs and resumes them.

B. Checkpoint/Restart and Fault Tolerance

It is often difficult for modern applications that run for long durations on large machines to get the allocations they need at one time. Hence, checkpointing the state of the application to disk for split execution is a common requirement. In such scenarios, it is an annoying constraint to require the same core count for executing a multiple phase application run every time the execution is resumed. These difficulties are exacerbated if faults, e.g. failure of a node, are observed during job execution. With faults becoming more frequent as the system sizes grow,

being able to continue execution without having to restart from a disk based checkpoint is becoming an essential feature required of modern parallel frameworks.

Over-decomposition based migratability provides a simple but powerful solution to these problems. Split execution is trivially enabled by migrating the work/data units to disk. Since the checkpoints are unit-based, restarting from them does not depend on the core counts. Tolerance to faults can be achieved by storing a copy of the units on a PE other than the PE they are mapped to. If one of the PE fails, the work/data units residing on it can be restarted on any other PE by using the stored copy.

Checkpointing Application State: CHARM++ uses the *PUP* framework (§II-D) to provide automated support for checkpointing the application state to disk. Using this chare-based checkpointing, the application can be restarted on any number of PEs irrespective of the number of PEs in the original run. To perform the checkpointing, the user only needs to call CkStartCheckpoint("log",callback), where "log" is the path to the checkpoints. `callback` is called when the checkpoint (or restart) is complete. To restart from checkpoints, one simply needs to pass a runtime argument *+restart log*.

Tolerating Process Failures: CHARM++ offers a double in-memory fault tolerance mechanism for applications running on unreliable systems [7]. In this scheme, periodically, two copies of checkpoints are stored: one in the local memory of the PE and the other in the memory of a remote PE. On failure, a new PE replaces the crashed PE, and the most recent checkpoints of chares running on the old PEs are copied to it. Thereafter, every PE rolls back to the last checkpoint, and the application continues to make progress. The fault tolerance mechanism provided by CHARM++ can be used by calling CkStartMemCheckpoint(callback), where `callback` is called when the checkpoint(or restart) is complete. The in-memory based checkpointing algorithm helps reducing the checkpoint overhead in comparison to the file system based checkpointing scheme by storing the checkpoint in memory.

C. Power Awareness

The power and energy budget is becoming more important for future exascale machines and studies have shown that cooling energy comprises of 40%-50% of the energy consumed by a data center [8]. To reduce this, the computer room air conditioning (CRAC) temperature can be set to a higher degree. However, higher room temperature can cause overheating of cores which reduces hardware reliability [9]. Dynamic Voltage Frequency Scaling (DVFS) is commonly used to prevent overheating by modulating chip frequency and voltage. However, changing the frequency of the chips can result in load imbalance and increase the execution time. Therefore, it becomes critical to perform load balancing along with DVFS.

The CHARM++ scheme leaves it to the RTS to adaptively control the chip temperature using DVFS. Over-decomposition and migratability are critical features required to ensure savings in cooling energy using this technique. The RTS monitors individual chip temperatures periodically and uses DVFS to constrain the chip temperature to a specified threshold. It decreases the frequency of hot cores and increases the ones that are well below the threshold. This can introduce significant

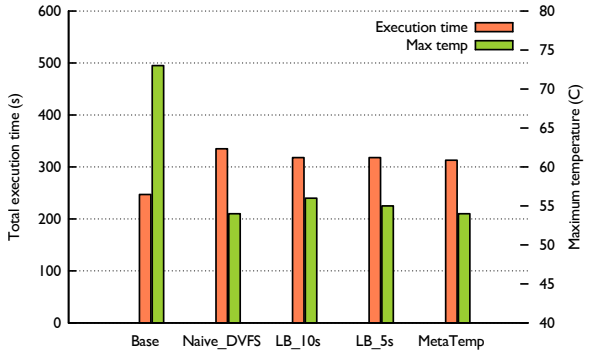


Fig. 4: The adaptive RTS successfully reduces the timing penalty while restraining the core temperatures.

amount of load imbalance in a tightly coupled application. To mitigate the load imbalance, the RTS monitors the application characteristics and whenever it detects load imbalance, it triggers a call to the load balancer [10].

When assigning tasks to a core, the load balancer scales the load according to the frequency of the core to ensure good load balance even in heterogeneous conditions. We have shown that this scheme can reduce the cooling energy considerably in comparison to naive DVFS [11]. The benefits can be seen in Figure 4 where the RTS is able to reduce the timing penalty and restrain core temperatures. Here, CRAC is set to 74°F and the threshold temperature is 50°C. *Base* case shows the application run without any modifications. *Naive_DVFS* applies DVFS periodically, while *LB_10s* and *LB_5s* perform load balancing along with DVFS every 10 and 5 seconds respectively. *MetaTemp* does load balancing whenever the benefit outweighs the cost.

D. Malleability

Malleability is the ability of parallel jobs to shrink or expand the number of processors on which they are executing at run-time in response to an external command. A parallel RTS which can render applications malleable, when used in conjunction with intelligent adaptive job scheduling algorithms can significantly improve a) cluster utilization, e.g, by expanding the running jobs when cluster demand is low, and b) average job response time, e.g. by shrinking the running jobs when cluster demand is high [12]–[16].

A processor-centric programming model makes shrinking or expanding extremely challenging since that necessitates too much application-specific programming effort for data redistribution after a shrink or expand event. In contrast, an object-centric over-decomposed system provides an elegant solution: automatic RTS-managed remapping of migratable objects to continuing/new processes after a shrink/expand event.

Using the capabilities mentioned above, CHARM++ jobs have the ability to shrink or expand by invoking a customized load balancer. The main idea is to evacuate chares from nodes which would be removed in case of shrink, and provide them to the new processes in case of expand [16]. However, chare redistribution by itself is insufficient since that leaves residual processes in case of shrink. These processes acts as processor-level agents which can interfere with other user's processes. The CHARM++ team recently enhanced the shrink/expand capabilities eliminating the need of these residual processors.

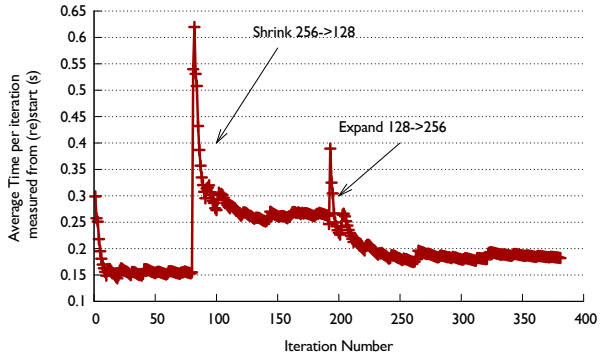


Fig. 5: Adapting load distribution on shrink and expand

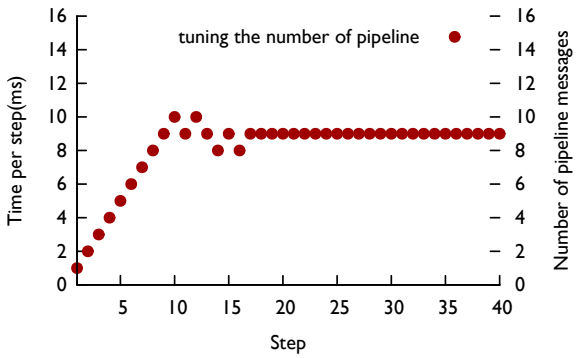


Fig. 6: Tuning the number of pipelines in a ping benchmark

Figure 5 demonstrates the shrink/expand capability using LeanMD. Initially the application is running on 256 cores of Stampede supercomputer at TACC. On a shrink request (sent through CHARM++ CCS [17] mechanism), the RTS reconfigures itself and application continues running on 128 cores. The iteration time doubles as expected. Later on, the number of cores is expanded back to 256, and the iteration time reduces accordingly. The peaks at shrink and expand denote the load balancing time. The total time for shrinking from 256 to 128 is 2.7s and the time for expanding from 128 to 256 is 7.2s. The time is dominated by the time taken to restart the application processes and reconnect them using a start-up protocol.

E. Introspective Control System

With the increasing complexity of applications and hardware, many researchers are actively working on runtime systems to support their programming models efficiently. For example, communication optimization in Partitioned Global Address Space (PGAS) is supported by the GASNET RTS [18]. Cilk is designed as an efficient multithread RTS by incorporating the work stealing scheduler [19], [20]. The new languages, Chapel [21] and X10 [22], both have their own powerful runtime systems.

CHARM++ provides an introspective adaptive RTS which automates task mapping, message scheduling, memory management and low level communication. Due to these capabilities, the RTS has the full knowledge of application behaviors, and can possibly reconfigure them for performance. In addition to configurable load balancing, we have also developed a more generic scheme based on an introspective control system to *reconfigure* both the RTS and the applications. This framework provides an easy-to-use interface for applications to interact with the control system by *control points* [23], [24]. Control points are tunable parameters with information about the expected effects of changing the parameters. The control system monitors the application events, collects performance data and performs automatic analysis to detect performance bottlenecks. Based on the performance results and a set of expert knowledge rules, the control system makes decisions about what control points to adjust and how to adjust them. The new configuration is then fed back to the application and RTS.

After studying various benchmarks, mini-apps and real world applications, we have identified various common control points for applications and runtime systems. Some of the more commonly used ones are the block size in stencil code, parallel threshold for state space search problems, stages in pipeline communication, topology aware mapping scheme, fault tolerance frequency, load balancing strategy and frequency, etc. The programmer can also register their own control points to tune their applications. Figure 6 illustrates the process of changing the number of pipeline stages in a ping benchmark and how it affects the performance. Our control system is able to find the optimal value and stabilize the performance.

F. Communication Optimization for Fine-grained Messages

It is often convenient to use fine-grained messages to implement the interactions between objects in a CHARM++ program. Fine-grained communication can arise as a result of object-based decomposition, as small work and data units tend to produce small message payloads. For constant-sized problems, decomposition into fine-grained units may even be necessary in order to create sufficiently many objects to utilize the large number of processing elements on modern supercomputing systems. Messages used for synchronization, data requests, and orchestration, both at the application and RTS level, are also frequently characterized by small payload.

The utility of fine-grained communication in directly representing many communication scenarios is in contrast to its generally poor performance characteristics. Much of the per-message overhead at the application, RTS, and network level is independent of message size. In applications with high fine-grained message volume, this overhead can dominate run time.

The CHARM++ Topological Routing and Aggregation Module (TRAM) is a library that improves fine-grained communication performance by coalescing fine-grained communication units, or *data items*, into larger messages. TRAM constructs a virtual topology comprising the processes in the parallel run, and defines *peers* within this topology as any processes that can be reached by traveling an arbitrary number of hops along a single dimension. Separately aggregating data items for each destination requires substantial memory and limits the potential for aggregating concurrent data items with different destinations but common sub-paths. To address this issue, TRAM aggregates data items at the level of peers. Items whose destinations are not in the set of peers at the source process will be routed through one or more intermediate destinations along a minimal route. As a result, the memory footprint of the buffer space will normally fit in lower level cache.

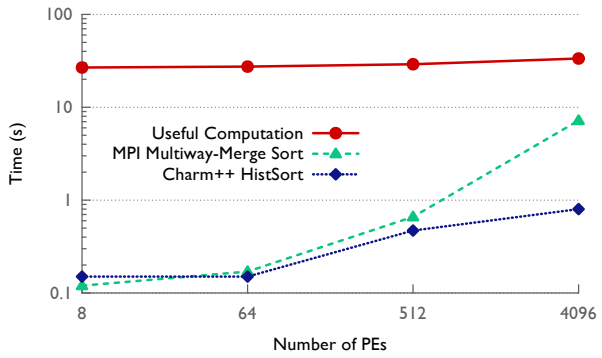


Fig. 7: CHARM: Interoperation removes scaling bottleneck.

G. Interoperation

Many modern applications consist of multiple modules (or phases) that are more suitable to different parallel programming paradigms. For optimal performance and productivity, it is critical that the programmer is allowed to implement a module in the language that suits it the most. For example, collision detection is required in many applications that may be implemented in MPI. However, given the asynchronous communication and load balancing required for scalable collision detections, CHARM++ is suitable for implementing it [25].

Effortless execution of the modules developed in different language is enabled by *interoperation*, which we believe is an important feature for parallel languages. To this end, CHARM++ provides a simple interface that makes its modules invocable from other languages as external libraries. This interface is currently used to enable interoperation with MPI. Any CHARM++ module can be made invocable from another language by adding an interface function to it. In a manner similar to a regular library invocation, the interface function is used to exchange data and transfer control to CHARM++. A typical interface function activates the RTS and begins execution of the module. On completion, the RTS returns control back to the interface function, which in turns returns it to its calling program. An MPI program that invokes a CHARM++ module is required to initialize CHARM++ by calling CharmLibInit before any CHARM++ module invocation.

We use CHARM [26], a production cosmological and astrophysical code implemented in MPI, to demonstrate the practical utility of interoperation between MPI and CHARM++. In order to remove the load imbalance caused by non-uniform particle distribution, CHARM performs a global sorting operation that redistributes the particles in every step before the computation is performed. As shown in Figure 7, the global sorting operation implemented in MPI becomes a performance bottleneck on large core counts (23% of the total time is spent in sorting at 4096 cores). While one can possibly implement a scalable sorting library in MPI, the features required by sorting operation — asynchronous and unexpected messages — suits CHARM++ more. As a result, a highly scalable sorting library exists in CHARM++ [27]. Enabled by interoperation, we offloaded the global sorting operation in CHARM to CHARM++'s sorting library, which removed the scalability bottleneck — 2% of the total time is spent in sorting at 4096 cores (Figure 7).

IV. PERFORMANCE BENCHMARKS

Next, we demonstrate the practical applicability of the RTS features (Section IV) in mini-applications and real applications. We also present how the RTS can be useful for HPC in cloud.

A. Adaptive Mesh Refinement

Adaptive Mesh Refinement [28] is used widely for obtaining numerical solution of partial differential equations. Its applications span a diverse set of fields such as computational fluid dynamics, astrophysics, etc. In numerical simulations using AMR, only those regions of the mesh that need higher resolution are refined to a higher depth, while others are kept at a coarser refinement level. As the simulation progresses, refinement level of the mesh regions is dynamically updated every few iterations.

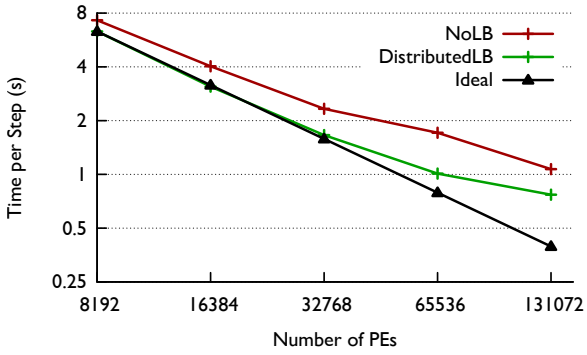
AMR3D is a mini-app written in CHARM++ for tree-based structured adaptive mesh refinement (SAMR) simulations. It performs a 3D finite-difference simulation of advection, which is a first-order upwind method. AMR3D is a very rich application that benefits from several CHARM++ features:

1) Object Based Decomposition: CHARM++'s object oriented approach significantly simplifies the expression of the program logic as a block, which is the basic unit of computation in the mesh. Unlike the process-centric programming languages such as MPI, the programmer has to write code to explicitly manage multiple blocks (possibly at different refinement levels) in a process. Block-based expression makes it very productive for the programmer to implement AMR codes [29], which are known to be very tedious to manage, maintain, and debug. Additionally, object based decomposition helps to overlap communication of one block with computation of another block on the same process.

In our implementation, we represent blocks as a chare array with custom array index. During mesh restructuring, new blocks can be dynamically inserted to or deleted from the chare array. We use bit vector indices that correspond to location of the block in the oct-tree. With bit vector indexing, a chare can find the index of its parent and neighbors by simple local operations on its own index. The RTS can redistribute the blocks without any change to the logic.

2) Dynamic Distributed Load Balancing: AMR requires dynamic load balancing because frequent refining and coarsening of the mesh grid creates load imbalance across processes. Figure 8 (left) shows the strong-scaling performance of AMR3D on $8K$ to $128K$ processes (4 processes/core) of BG/Q achieving 46% parallel efficiency with load balancing. We use a distributed load balancing strategy [30] to achieve significant performance benefit of 40% at $128K$ processes over the run without any load balancing.

3) In-memory Checkpoint/Restart (with simulated failure): Figure 8 (right) shows checkpoint and restart time of AMR3D with dynamic depth range from 2 to 9 on BG/Q. As the number of processes increase from $2K$ to $32K$, checkpoint size per core decreases. Therefore, checkpoint time decreases from 394ms on $2K$ processes to 29ms on $32K$ processes. The restart time also decreases from 2.24s at $2K$ processes to 470ms at $32K$ processes.



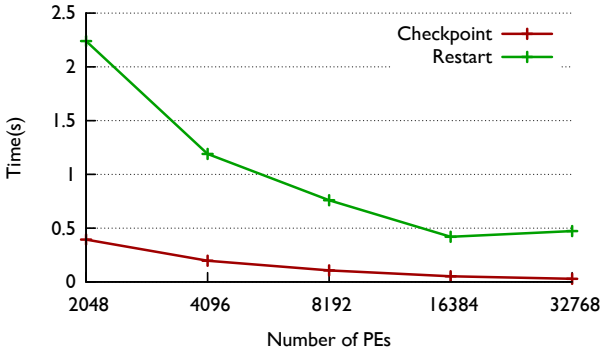


Fig. 8: AMR3D with dynamic depth range from 2 to 9, on Vesta (IBM BG/Q)

4) Overcoming Typical Bottlenecks in AMR codes: Typical AMR implementations such as Enzo [31], Chombo [32], Flash [33], etc. require the tree structure of the mesh (or all the patch information in patch-based simulations) to be replicated on each process to facilitate mesh restructuring, requiring $O(\#blocks)$ memory per process. This acts as a memory bottleneck. The distributed object location manager and bit vector indexing in CHARM++ eliminates this memory bottleneck, requiring only $O(\#blocks/P)$ memory per process. The scalable quiescence detection feature in RTS is used during mesh restructuring to determine when all refinement decisions are done. Therefore, mesh restructuring requires just O(1) global collective call. On the contrary, typical AMR implementations take O(d) global collective calls for mesh restructuring (where d is the depth of the tree). This approach makes AMR3D highly scalable. Details of the algorithms can be found in [34].

B. Molecular Dynamics

LeanMD is a molecular dynamics simulation mini-app written in Charm++. It simulates the behavior of atoms based on Lennard-Jones potential, where the force calculations on atoms are performed within a cut-off distance. The computation of this mini-app is representative of the non-bonded force calculations in NAMD, which take a large fraction of the wall clock time. NAMD is an application written in Charm++ that won the 2002 Gordon Bell award. We present some of the Charm++ features that are leveraged to improve performance.

1) Over-Decomposition: The 3D simulation space consisting of atoms is decomposed into cells, to form a dense $3D$ chare array called *Cells*. In one iteration of the simulation, force calculations are performed for all pairs of atoms in a sparse $6D$ chare array called *Computes*, which consumes most of the simulation time. Processor based decomposition of *Computes* would have resulted in smaller number of *Computes* with imbalanced distribution of load. Over-decomposition of *Computes*, which enables overlap of communication and computation as well as load balancing, is critical to achieve scalable performance for LeanMD (and NAMD).

2) Adaptive Load Balancing: For LeanMD , load balancing is critical to obtain good performance. The load of a *Compute* element is proportional to the number of atoms in the cells for which it is computing the forces. The RTS monitors the application continuously and invokes the load balancer when an imbalance is detected and if the cost of load balancing is not more than the benefit. Figure 9 shows the scaling of LeanMD for a 2.8 million atoms system on $1K$ cores

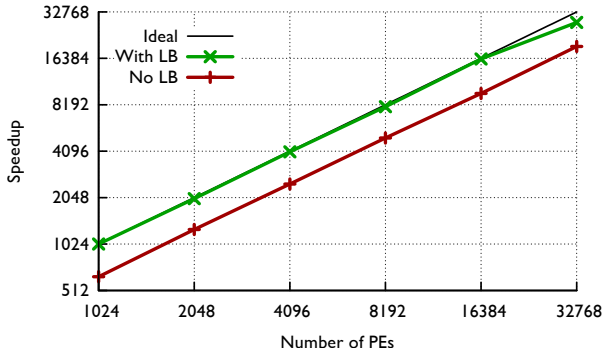


Fig. 9: LeanMD performance on Vesta (IBM BG/Q)

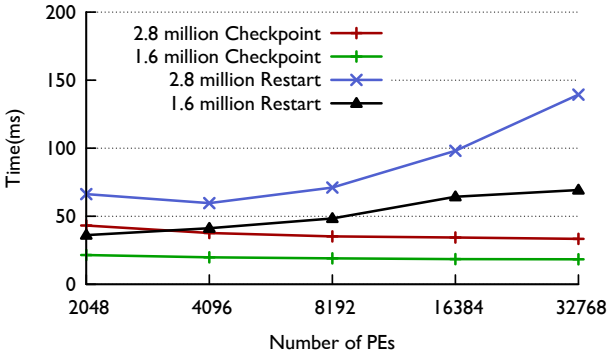


Fig. 10: LeanMD Checkpoint and restart on Vesta (IBM BG/Q)

to $32K$ cores of Vesta, an IBM Blue Gene/Q. We obtain good scaling performance: at $32K$ cores, the time per step is 44 ms/step. Note that the use of scalable hierarchical load balancer, *HybridLB*, improves the performance by at least 40%.

3) In-memory Checkpoint/Restart (with simulated failure): Figure 10 shows the checkpoint and the restart time of LeanMD for a 2.8 million atoms system and a 1.6 million atoms system on a BlueGene/Q. Compared to AMR, the checkpoint size of LeanMD is much smaller. Thus the checkpoint and restart times for LeanMD are both in few tens of milliseconds range. When the number of processors increases from $2K$ to $32K$, the checkpoint time for the 2.8 million atom system decreases from 43ms to 33ms. During restart, several barriers are used to ensure consistency until the crashed node is recovered. The restart time for the 2.8 million atom system increases slightly from 66 ms on $4K$ processors to 139 ms on $32K$ processors due to the effect of barriers.

As a full-fledged molecular dynamics application, we show

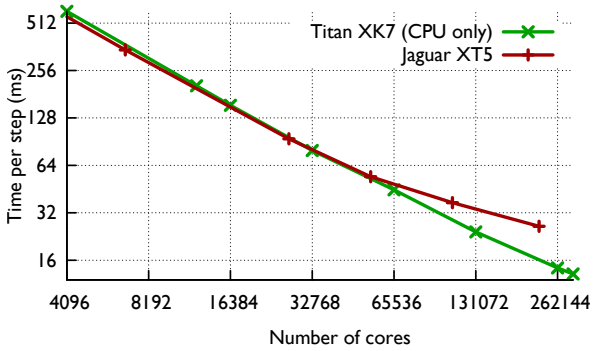


Fig. 11: NAMD strong scaling for 100 million atom benchmark

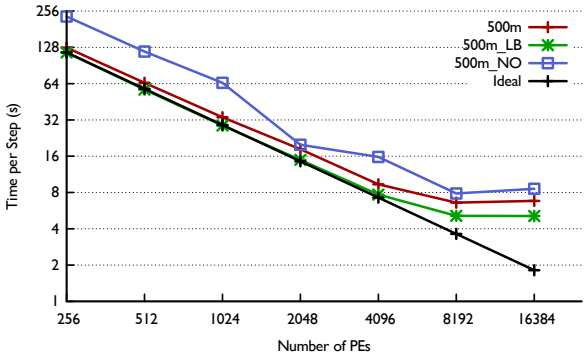


Fig. 12: Barnes-Hut Performance on Blue Waters (Cray XE6)

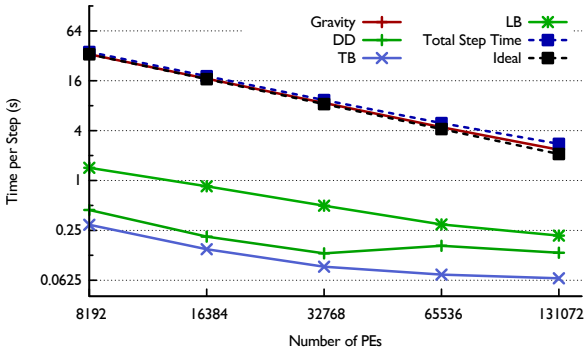


Fig. 13: Performance of ChaNGa cosmo25 dataset evolving 2 billion particles on Blue Waters (Cray XE6)

the results for NAMD running on various major supercomputers. Figure 11 shows its strong scaling for the 100 million atom benchmarks, where is scaled up to the full Titan XK7 and Jaguar XT5 systems. This demonstrates the capability of CHARM++ programming model to scale real applications.

C. Barnes Hut

Barnes-Hut is an N-body simulation algorithm which simulates N moving particles (masses) under the influence of gravitational forces [35]. At each discrete time step, the net forces are calculated on each particle, then the positions and velocities of the particles are updated. Barnes-Hut implementation and performance relies on the following CHARM++ features:

1) Object Based Decomposition: The $3D$ space is over-decomposed into a chare array called $TreePieces$ using an oct decomposition. Over-decomposition enables the RTS to map multiple $TreePiece$ elements to a PE enabling load balancing and overlap of communication and computation. Figure 12 shows the performance benefits – 40% (500m) compared to the case with one object per process (500m_NO).

2) Prioritized Messages: CHARM++ allows associating priorities with messages (method invocations). Prioritization is used in Barnes-Hut to give precedence to remote data requests. Since the remote requests might take longer than the local computation, they are given a higher priority.

3) Adaptive Load Balancing: Uneven distribution of the particles creates load imbalance. We use a load balancing strategy which performs Orthogonal Recursive Bisection (ORB) to balance the load among PEs. Figure 12 shows the scaling of this Barnes-Hut mini-app for 500 million particles. We obtain good performance due to over-decomposition and load balancing with a time per step of 5 seconds at 8192 cores.

Figure 13 shows the scaling of ChaNGa, a cosmology application in CHARM++, from $8K$ to $128K$ cores on Blue Waters evolving 2 billion particles. It also shows the break down of the total execution time in terms of other phases such as Gravity, DD (Domain Decomposition), TB (Tree Build) and LB (Load Balancing). We obtain good performance scaling and at $128K$ cores the Total Step Time takes a total of 2.7 seconds with a parallel efficiency of 80% with respect to $8K$ cores.

D. AMPI Example with Lulesh

Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) is a widely used mini-app written in MPI [36], [37]. It mimics the behavior of a large class of science and engineering applications that use hydrodynamics models. LULESH solves the hydrodynamics equations of state on an unstructured, hexahedral mesh. The length of each time step is calculated dynamically, and the communication pattern includes both nearest neighbor and global communication. Typical hydrodynamics applications exhibit load imbalance due to the varying amounts of computation for different materials. LULESH models this imbalance by having disjoint regions in the simulation domain. However, the load imbalance in LULESH is designed to be small.

1) Porting to AMPI: Porting LULESH to AMPI requires minimal effort since it is a modular C++ program, with very few global and static variables. It can run using AMPI by merely adding the appropriate compiler and linker flags. To use other features of the CHARM++ framework, *MPI_Migrate()* calls need to be added to the main iteration loop. Although packing the user's data for migration can be automated using *iso-malloc*, we wrote a *pup* routine for better portability. This is straightforward for LULESH since the application's data is encapsulated in the *Domain* class.

2) Improving Cache Utilization using AMPI: Cache optimization is very important for the performance of LULESH since it runs multiple kernels on the simulated domain in each iteration. However, applying cache optimizations such as blocking (tiling) manually can be challenging for legacy MPI codes. In particular, the memory access pattern of LULESH is complex because it uses an unstructured mesh (with non-unit stride accesses) and multiple domain regions.

Virtualization presents the illusion of more PEs to the application, which results in less data per virtual PE for a fixed input size. This improves cache efficiency if the smaller data of each virtual PE fits in the cache. For LULESH, we

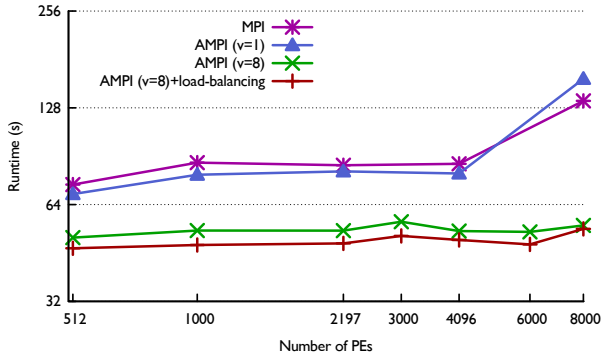


Fig. 14: LULESH performance comparison: MPI vs. AMPI for automatic cache optimization and load balancing on Hopper

observe that each Hopper node is working on about 283MB of data. The total size of L2 and L3 caches on each node of Hopper is about 36MB. With eight-way virtualization, we decrease the active working set size to about 35MB, which is below the total cache size. Effectively, each iteration's work is performed in eight portions, each with smaller working sets. Figure 14 presents the weak scaling of LULESH on native MPI and AMPI. The domain size is the default (27000 elements per PE). The AMPI run with a virtualization ratio of eight provides a 2.4x speedup with the same source code, simply by reducing the size of the working set in the cache.

3) Automatic Load Balancing using AMPI: Figure 14 also demonstrates that AMPI can alleviate the small load imbalance of LULESH automatically. This is done by migrating the virtual PEs from overloaded processors.

4) Running on Various Number of Cores: Normally LULESH requires a cubic number of cores, which is a major limitation for users. Virtualization allows the user to run on any number of cores by providing a cubic number of virtual processors. To illustrate this, Figure 14 presents data for non-cubic core counts (3000 and 6000). It can also be seen that there is no major performance overhead associated with this feature.

E. PDES

In parallel discrete event simulation (PDES), logical processes (LPs) execute discrete events in order to simulate a scenario or physical phenomenon written by a simulation designer. Events are timestamped with a virtual time and to run correctly, events must be executed in non-decreasing timestamp order. This ordering imposed on events is what makes PDES applications difficult to scale.

We have written a PDES mini-app in CHARM++ using the YAWNS protocol, which is an example of a windowed conservative protocol [38]. A program using the YAWNS protocol is in one of two phases: window calculation or event execution. During the window calculation phase, each LP determines the earliest possible time it can create a new event. Then, the minimum of these times is found using a reduction. This determines a window in which events can be executed without being preempted by a new event. The execution phase executes all events in the current window.

We benchmarked our mini-app using the PHOLD simulation benchmark. In the PHOLD benchmark, each LP generates an initial load of events, each with a random duration and destination. To process an event, an LP computes a new random duration for the event, and forwards it to a new, randomly determined LP.

The PDES mini-app relies on the following CHARM++ features in order to operate efficiently:

1) Over-decomposition: During the execution phase of the YAWNS protocol, only LPs which have events in the current window have useful work to do. Because CHARM++ utilizes over-decomposition, when one LP doesn't have work in the current execution window, the PE can choose another LP that does have work in the current window, thus minimizing idle time on the PE. Figure 15a shows the effects of over-decomposition by comparing event rates as we increase the number of LPs per PE from 64 to 256, with a fixed initial event load of 32 events per LP.

2) Asynchronous Message-Driven Execution: In many PDES applications, the communication patterns between LPs cannot be determined *a priori*. Recipients have no way of knowing when and from where they will receive new events, and senders have no way of knowing what other events will need to be executed by the recipient. In a synchronous communication model this would be complex to code and prone to deadlocks. The message driven, asynchronous nature of CHARM++ eliminates much of this complexity. Recipient LPs are executed by the RTS when messages arrive for them, removing the need for them to explicitly receive events, and senders can immediately continue execution after a send rather than waiting on the recipient.

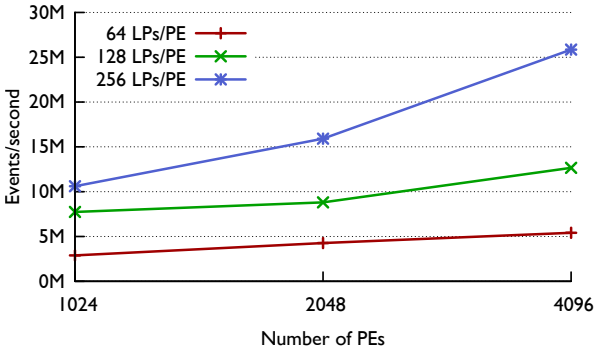
3) TRAM: A common characteristic of PDES applications is a very high number of fine grained events. This holds true in our benchmark, allowing us to leverage the benefits of TRAM. In Figure, 15b we show the impact of TRAM in runs with 256 LPs per PE, and initial event loads of 64 and 1024 events per LP. At low communication volumes, aggregation of messages increases average message latency, making the direct send approach faster. We see this for the case of 64 messages per LP on 1024 PEs. For higher communication volumes, using TRAM generally led to better performance than direct sending, reaching a peak of over 50 million events per second with 1024 events per LP on 4096 PEs.

F. Beyond Supercomputers: HPC in Cloud

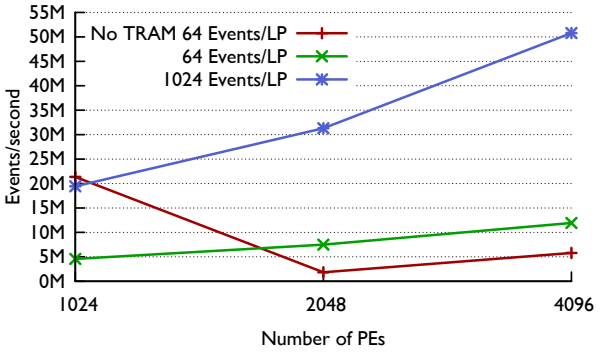
Cloud computing with Infrastructure-as-a-Service offerings (such as Amazon EC2 [39]) has recently emerged as a promising addition to traditional supercomputers. Clouds are especially attractive to small and medium scale organizations, especially those with emerging or sporadic HPC demands, since they can benefit from the pay-as-you-go model (renting vs. buying) and elasticity – on-demand provisioning in clouds.

However, despite this potential to spread the outreach of HPC, there is a mismatch between current cloud environments and typical HPC requirements. Poor interconnect and I/O performance, HPC-agnostic cloud schedulers, and the inherent heterogeneity and multi-tenancy are some of the bottlenecks for HPC in the cloud [40]–[45].

Our research in the HPC-cloud domain has shown that the following features of over-decomposed adaptive HPC runtimes can be pivotal in meeting these challenges [46]. We present our findings on a private cloud comprising 8 Intel Xeon



(a) Varying LPs per PE with 32 events per LP



(b) Varying events per LP with 256 LPs per PE

Fig. 15: Weak scaling runs of the PHOLD benchmark on Stampede. In (a) we increase the number of LPs per PE with a fixed initial event load of 32 events per LP. In (b) we show the benefits of TRAM at high event counts with 256 LPs per PE.

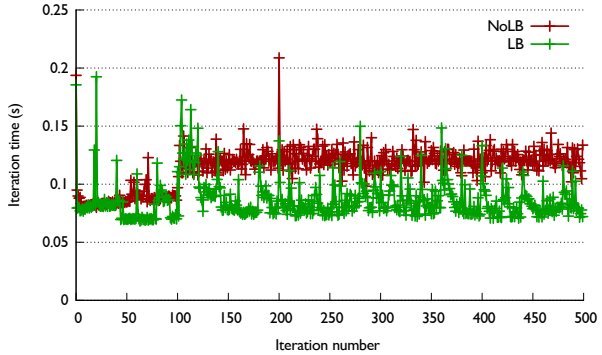


Fig. 16: Stencil2D's performance with load balancing in cloud

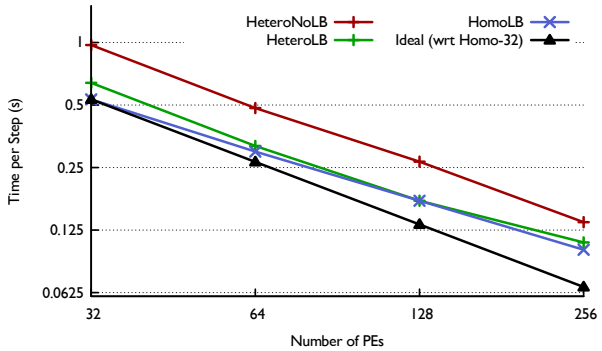


Fig. 17: Performance and scalability of LeanMD in heterogeneous cloud environments

X5650 @2.67GHz nodes connected using 1Gig Ethernet, and virtualized using kvm [47].

1) Over-decomposition: Latency and bandwidth microbenchmarks showed that the underlying network in most clouds (typically commodity Ethernet) performs an order of magnitude worse compared to typical HPC interconnects [44], [45]. Automatic overlap of computation and communication, enabled by over-decomposition, minimizes the effect of slow network on application performance. In our experiments, the iteration time for Stencil2D (grid size 4k by 4k) on 32 VMs improved by 2.4X – from 77ms with 1 chare/process to 32ms with 8 chares/process.

2) Dynamic Load Balancing: In clouds there is a) static heterogeneity, i.e., underlying physical nodes may be different, and b) dynamic heterogeneity which is an artifact of the interference arising from multi-tenancy, i.e. multiple VMs from multiple users may be sharing/entering/exiting the same physical machine. The rich CHARM++ RTS facilitates the development of novel load balancing techniques to achieve good performance in dynamic and heterogeneous cloud environments. The main idea is to (1) infer whether the load imbalance is application intrinsic or caused by extraneous factors such as interference and (2) dynamically redistribute chares from overloaded to underloaded VMs [46].

Figure 16 shows iteration time for Stencil2D on 32 VMs (8 nodes). Here, an interfering VM was started on one node after 100 iterations. The difference between the two curves after 100 iterations illustrates the benefits of heterogeneity-aware load balancing. Load balancing happens every 20 steps, which also manifests itself as a spike in the iteration time.

Another challenge in clouds is that the interference is unpredictable, and varies from run to run – making it impossible to *a priori* determine optimal load balancing period. Hence, instead of application-triggered periodic load balancing, we switch to an RTS-triggered approach [48]. To evaluate this approach on a larger scale, we used a cloud setup on Grid'5000 [49] testbed – Graphene cluster at Nancy site, with Linux containers [50] for virtualization and Distem [51] for creation of artificial heterogeneity in homogeneous clusters. We introduced heterogeneity by making one node's effective CPU frequency as 0.7X. Figure 17 shows LeanMD's performance and scalability on this setup. Performance attained with heterogeneous-awareness is close to the homogeneous case.

V. CONCLUSION

In this paper, we describe key concepts that are essential for parallel applications to take advantage of today's modern supercomputers. We show the benefits of over-decomposition, message-driven execution, and migration both by describing the powerful features they enable and showing empirical results from a breadth of benchmark mini-apps and full-scale applications written in CHARM++. CHARM++ handles the features described in this paper in a way that is transparent to the programmer, allowing users of CHARM++ to focus on the details of their application while the adaptive RTS handles the details of efficient parallel execution. As machines continue to grow in size, these parallel framework features will become even more important for achieving scalable performance.

REFERENCES

[1] D. Brown et al., "Scientific Grand Challenges: Crosscutting Technologies for Computing at the Exascale." U.S. DOE PNNL 20168, Report from Workshop on Feb. 2-4, 2010, Washington, DC, Tech. Rep., 2011.

[2] "Top Ten Exascale Research Challenges." U.S. DOE, Report from DOE ASCAC Subcommittee, Tech. Rep., 2014, http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf.

[3] "Open Community Runtime," http://01.org/projects/open-community-runtime.

[4] L. Kale, A. Arya, A. Bhatele, A. Gupta, N. Jain, P. Jetley, J. Lifflander, P. Miller, Y. Sun, R. Venkataraman, L. Wesolowski, and G. Zheng, "Charm++ for productivity and performance: A submission to the 2011 HPC class II challenge," Parallel Programming Laboratory, Tech. Rep. 11-49, November 2011.

[5] O. S. Lawlor and L. V. Kalé, "Supporting dynamic parallel object arrays," *Concurrency and Computation: Practice and Experience*, vol. 15, pp. 371–393, 2003.

[6] G. Zheng, "Achieving high performance on extremely large parallel machines: performance prediction and load balancing," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[7] G. Zheng, L. Shi, and L. V. Kalé, "FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI," in *2004 IEEE Cluster*, San Diego, CA, September 2004, pp. 93–103.

[8] R. Sawyer, "Calculating total power requirements for data centers," *White Paper, American Power Conversion*, 2004.

[9] O. Sarood, E. Meneses, and L. Kalé, "A 'cool' way of improving the reliability of hpc machines," in *Proceedings of The International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, CO, USA, November 2013.

[10] H. Menon, B. Acun, S. G. De Gonzalo, O. Sarood, and L. Kalé, "Thermal aware automated load balancing for hpc applications," in *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*. IEEE, 2013, pp. 1–8.

[11] O. Sarood and L. V. Kalé, "A 'cool' load balancer for parallel applications," in *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*, Seattle, WA, November 2011.

[12] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong, "Theory and practice in parallel job scheduling," in *Proceedings of the Job Scheduling Strategies for Parallel Processing*, ser. IPPS '97. London, UK, UK: Springer-Verlag, 1997, pp. 1–34. [Online]. Available: http://dl.acm.org/citation.cfm?id=646378.689517

[13] K. El Maghraoui, T. Desell, B. Szymanski, and C. Varela, "Dynamic malleability in iterative mpi applications," in *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, 2007, pp. 591–598.

[14] M. C. Cera, Y. Georgiou, O. Richard, N. Maillard, and P. O. A. Navaux, "Supporting malleability in parallel architectures with dynamic cpusets mapping and dynamic mpi," in *Proceedings of the 11th international conference on Distributed computing and networking*, ser. ICDCN'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 242–257. [Online]. Available: http://dl.acm.org/citation.cfm?id=2018057.2018090

[15] G. Utrera, J. Corbaln, J. Labarta, and D. D. D. C. (dac, "Implementing malleability on mpi jobs," in *Proc. 13 th Intl Conf. on Parallel Architecture and Compilation Techniques (PACT04*. IEEE Computer Society, 2004, pp. 215–224.

[16] L. V. Kalé, S. Kumar, and J. DeSouza, "A malleable-job system for timeshared parallel machines," in *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, May 2002.

[17] *The CONVERSE programming language manual*, Department of Computer Science,University of Illinois at Urbana-Champaign, Urbana, IL, 2006.

[18] "Gasnet: A portable high-performance communication layer for global address-space languages," 2002. [Online]. Available: http://gasnet.cs.berkeley.edu/

[19] 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.

[20] ——, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, 1996.

[21] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, pp. 291–312, August 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1286120.1286123

[22] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA*. New York, NY, USA: ACM, 2005, pp. 519–538.

[23] I. Dooley and L. V. Kale, "Control points for adaptive parallel performance tuning," November 2008.

[24] I. Dooley, "Intelligent runtime tuning of parallel applications with control points," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2010, http://charm.cs.uiuc.edu/papers/DooleyPhDThesis10.shtml.

[25] O. S. Lawlor and L. V. Kalé, "A voxel-based parallel collision detection algorithm," in *Proceedings of the International Conference in Supercomputing*. ACM Press, June 2002, pp. 285–293.

[26] F. Miniati and P. Colella, "Block structured adaptive mesh and time refinement for hybrid, hyperbolic+n-body systems," *J. Comput. Phys.*, vol. 227, no. 1, pp. 400–430, Nov. 2007.

[27] E. Solomonik and L. V. Kale, "Highly Scalable Parallel Sorting," in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2010.

[28] M. J. Berger and J. Oliger, "Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations," *Journal of computational Physics*, vol. 53, no. 3, pp. 484–512, 1984.

[29] L. Kale, A. Arya, N. Jain, A. Langer, J. Lifflander, H. Menon, X. Ni, Y. Sun, E. Totoni, R. Venkataraman, and L. Wesolowski, "Migratable objects + active messages + adaptive runtime = productivity + performance a submission to 2012 HPC class II challenge," Parallel Programming Laboratory, Tech. Rep. 12-47, November 2012.

[30] H. Menon and L. Kalé, "A distributed dynamic load balancer for iterative applications," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 15.

[31] B. Oshea, G. Bryan, J. Bordner, M. Norman, T. Abel, R. Harkness, and A. Kritsuk, "Introducing enzo, an amr cosmology application," in *Adaptive Mesh Refinement - Theory and Applications*, ser. Lecture Notes in Computational Science and Engineering. Springer Berlin Heidelberg, 2005, vol. 41, pp. 341–349.

[32] "Chombo Software Package for AMR Applications," http://seesar.lbl.gov/anag/chombo.

[33] G. Weirs, V. Dwarkadas, T. Plewa, C. Tomkins, and M. Marr-Lyon, "Validating the Flash code: vortex-dominated flows," in *Astrophysics and Space Science*. Springer, 2005, vol. 298, pp. 341–346.

[34] A. Langer, J. Lifflander, P. Miller, K.-C. Pan, , L. V. Kale, and P. Ricker, "Scalable Algorithms for Distributed-Memory Adaptive Mesh Refinement," in *Proceedings of the 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2012). To Appear*, New York, USA, October 2012.

[35] J. Barnes and P. Hut, "A hierarchical $O(N \log N)$ force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, December 1986.

[36] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[37] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, August 2013.

[38] P. M. Dickens, D. M. Nicol, P. F. Reynolds, Jr., and J. M. Duva, "Analysis of bounded time warp and comparison with yawns," *ACM Transactions on Modeling and Computer Simulation*, vol. 6, no. 4, pp. 297–320, Oct. 1996.

[39] "Amazon Elastic Compute Cloud (Amazon EC2)," http://aws.amazon.com/ec2.

[40] E. Walker, "Benchmarking Amazon EC2 for high-performance scientific computing," *LOGIN*, pp. 18–23, 2008.

[41] C. Evangelinos and C. N. Hill, "Cloud Computing for parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2." Cloud Computing and Its Applications, Oct. 2008.

[42] P. Mehrotra, J. Djomehri, S. Heistand, R. Hood, H. Jin, A. Lazanoff, S. Saini, and R. Biswas, "Performance Evaluation of Amazon EC2 for NASA HPC applications," in *Proceedings of the 3rd workshop on Scientific Cloud Computing*. New York, NY, USA: ACM, 2012.

[43] "Magellan Final Report," U.S. Department of Energy (DOE), Tech. Rep., 2011.

[44] A. Gupta and D. Milojicic, "Evaluation of hpc applications on cloud," in *Open Cirrus Summit (OCS), 2011 Sixth*, 2011, pp. 22–26.

[45] A. Gupta, L. V. Kalé, D. S. Milojicic, P. Faraboschi, R. Kaufmann, V. March, F. Gioachin, C. H. Suen, and B.-S. Lee, "The who, what, why and how of high performance computing applications in the cloud," in *Proceedings of the 5th IEEE International Conference on Cloud Computing Technology and Science*, ser. CloudCom '13, 2013.

[46] A. Gupta, O. Sarood, L. Kale, and D. Milojicic, "Improving hpc application performance in cloud through dynamic load balancing," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, 2013, pp. 402–409.

[47] "KVM – Kernel-based Virtual Machine," Redhat, Tech. Rep., 2009.

[48] H. Menon, N. Jain, G. Zheng, and L. V. Kalé, "Automated load balancing invocation based on application characteristics," in *IEEE Cluster 12*, Beijing, China, September 2012.

[49] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard, "Grid'5000: A large scale and highly reconfigurable grid experimental testbed," in *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, ser. GRID '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 99–106. [Online]. Available: http://dx.doi.org/10.1109/GRID.2005.1542730

[50] D. Schauer et al., "Linux containers version 0.7.0," June 2010, http://lxc.sourceforge.net/.

[51] L. Sarzyniec, T. Buchert, E. Jeanvoine, and L. Nussbaum, "Design and evaluation of a virtual experimental environment for distributed systems," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, Feb 2013, pp. 172–179.