Contents

XCharj: An Exascale Language with a Scalable Adaptive Runtime System

Applicant/University: Laxmikant V. Kalé, University of Illinois at Urbana-Champaign.
Street Address/City/State/Zip: 201 North Goodwin Avenue, Urbana, IL, 61801-2302
Principal Investigator: Laxmikant V. Kalé.
Postal Address: 201 North Goodwin Avenue, Urbana, IL, 61801-2302
Telephone Number: (217) 244-0094
Email: kale@illinois.edu
Funding Opportunity Announcement Number: DE-FOA-0000619.
DOE/Office of Science Program Office: Office of Advanced Scientific Computing Research (ASCR).
DOE/Office of Science Program Office Technical Contact: Dr. Sonia R. Sachs.

2012 X-Stack	**Year 1**	**Year 2**	**Year 3**	**Total**
Laxmikant Kale, UIUC	$400,000	$400,000	$400,000	$1,200,000
Collaborating Institutions				**Total**
Daniel Quinlan, LLNL	$250,000	$257,500	$265,225	$772,725
Robert Harrison, ORNL	$200,000	$200,000	$200,000	$600,000
Totals	**$850,000**	**$857,500**	**$865,225**	**$2,572,725k**

Table 1: Budget Allocation

Project Objectives

The transition to exascale is expected to be much more challenging than that to petascale, in part because of a much larger increase in concurrency, heterogeneity in the form of accelerators and process variation, and increased importance of power/energy considerations. It is compounded further by increasing sophistication of CSE applications, and the need for strong scaling.

The proposed research aims at addressing these challenges by developing adaptive runtime techniques and higher level abstractions supported by static analysis, and by evaluating these with DOE-relevant *mini-apps*.

The first objective is to develop a highly scalable, introspection-driven adaptive runtime system to combat the variability of the exascale environment. Specifically, we plan to develop an execution model based on over-decomposition into logical work and data units, and an introspection system that continually monitors execution and triggers corrective strategies. We will develop several classes of adaptive strategies including (a) scalable strategies for dynamic load balancing, dealing with interconnection topology, handling intra-node processor hierarchy and utilizing diverse accelerator hardware, (b) energy and temperature control strategies, and (c) fault tolerance strategies.

Raising the level of abstraction for application developers is another major objective of the proposed work. Our novel idea is to develop abstractions based on observed patterns of interaction between parallel entities. By narrowing focus to a particular class of interactions, we can create a programming dialect that is simpler, more analyzable, and less error-prone. We will develop multiple dialects, specialized to these patterns, and a general-purpose analyzable language that embeds them. Interoperability between them will be supported by virtue of our execution model.

A major objective is also to ensure that applications can transition to our stack easily, by a combination of supported conversion of MPI programs, and interoperability with MPI modules.

The project will draw upon application development expertise of the investigators to evaluate the techniques and abstractions developed, using mini-apps based on DOE applications.

Contents

1 Introduction

High-performance parallel computing has demonstrated the potential to engender breakthroughs in science and engineering in areas of national importance such energy, national security, and biomedicine. Although programming a large number of processors to solve a single problem faster has been a challenging task, scientists and engineers have persevered and delivered high-performing simulation codes for many application domains on machines with hundreds of teraFLOP/s (and in some cases, petaFLOP/s) performance.

However, it is now well accepted that the transition to the next level of performance, namely exaFLOP/s, would be substantially different and difficult. The evolution of semiconductor technology is at a point where clock speeds cannot be increased much further, and thermal considerations limit progress of processor design in significant ways. On the other hand, Moore's law regime is likely to continue for a decade, taking feature sizes down to 5-9 nm, and the number of transistors per chip over 50 billion. One can use these transistors to put hundreds of cores on each chip.

The first teraFLOP/s machine on the top 500 was Sandia's ASCI Red in 1997. It had 7,264 cores running at 200 MHz. The current top machine, Kei computer, is 10,000 times faster (10+PF), but has only 100 times more cores (705,024 cores in 88,128 chips). The remaining factor of 100 performance increase came from each core becoming faster. From here on, the per-core performance is likely to be fixed (or go lower, because of thermal considerations). This means a huge increase in degree of parallelism – with hundreds of millions (possibly billions) of hardware cores/threads – will be required to obtain exaFLOP/s performance.

Further, the cores will likely be heterogeneous: each node will likely consists of chips containing accelerator style processor cores as well as traditional processor cores optimized for single thread performance. The sheer number of components, as well as the semiconductor process technologies involved, will make it likely that the mean-time to component failure could be in minutes. Data movement will become the most expensive operation, both in terms of the execution time and the energy consumed. Total memory in the system is also expected to grow much slower than the compute power. So, memory per core (and per FLOP/s) will decrease considerably.

The **nature of exascale applications** also leads to new challenges:

- **Strong scaling** will be required at exascale for most applications, in part due to the memory size. Also, the problems of interest in many domains do not get much larger, but are required to be solved faster.
- Applications will get **more sophisticated**: For example, instead of using finer resolution everywhere when a larger machine becomes available, they will use *dynamic and adaptive refinements*. Also, strong scaling requirements lower the time per iteration/step, which increases the impact of load imbalances, which may be persistent or transient.
- Applications will tend to be **multi-module** and multi-physics in nature. There will be a need to cast expensive parallel software into modular yet efficient parallel libraries.

So, in spite of increasingly complex machine structures, we need to make development of CSE applications *easier* than it is now. This proposal addresses this challenge in a two-pronged manner:

1. **XARTS: Exascale Adaptive Runtime System** We first build upon an execution model based on the concepts of over-decomposition, message-driven execution, and macro-data-flow [1–3]. The ontology of this model naturally leads to empowering an adaptive runtime system that can continuously monitor execution — including machine variations and application evolution — at a granularity that allows it to intervene to optimize performance. XARTS will provide dynamic resource management, fault tolerance, communication optimizations, thermal/power management, and heterogeneity management, freeing the programmer and higher level languages from having to deal with these issues.

2. **XCharJ:** The API provided by XARTS components, with thin layers of support libraries, can be used to code applications directly. However, a compiler-supported higher level language can raise the level of abstractions further, increasing performance and producivity. One way to accomplish this is to provide support, including specialized syntax (dialect) and static analysis, for a few dominant *interaction patterns*. Compiler support will be aimed at basic optimizations [4, 5] targeted at these specialized notations,, rather than auto-parallelization or other complex strategies that have proved inadequate in the past.

Application Orientation: One of the guiding principle in our design is that all abstractions and features must be motivated by CSE applications, and they must be validated by use in multiple applications. While we develop and present our abstractions in a bottom-up manner, to ensure pragmatic implementability at each layer, this application orientation ensures that we develop concepts that will make an impact on the state of art of parallel programming for CSE. The PIs are exemplars of this approach to the science of parallel programming. The lead PIs' 1994 position paper explicitly elaborated this principle [6]. Harrison's development of abstraction in the MADNESS framework [7], Quinlan's ROSE framework [8], and Kale's Charm++/AMPI [1] were all driven by CSE applications, and have been used and validated in their use.

1.1 Broad Approach

The ideas in the proposal build upon much past work. Although exascale provides a qualitatively different challenge, the fundamental issues in parallelism still exist, albeit in a different mix. This makes it imperative that we build upon existing work, while seeking radical changes in programming models.

The work of lead PI (Kale) on Charm++ [1] (funded by DOE in part), and its adaptive runtime system is designed for dynamic and irregular context, and the concepts developed within it are highly relevant for exascale. It is by relying on these concepts (and in some cases, software modules) that we will be able to deliver research (and software) on an almost complete exascale software stack, with a relatively modest budget. However, the proposed work will need significantly more functionality than that is provided by Charm++, especially in scalable strategies, and in its higher level language.

The overall approach we take is shown in perspective in Figure 1. We now describe this broad approach and place it in the context of existing state-of-art work. We start with the lower levels, and build upwards.

A low level runtime system (LRTS) is needed at exascale to provide basic startup, communication and essential (minimal) OS services with a uniform API. We expect to leverage existing work in Charm++ RTS, and possibly that from the Unistack project, for this purpose. Work done on layers such as GASNET [9], ARMCI [10], MPICH [11], MVAPICH [12], and OpenMPI [13] is relevant for this purpose, and we hope to ally with those groups in an attempt to standardize this layer, and build upon it. The main attributes we need from this layer are efficient and flexible communication, and ability to withstand component failures. Although the ability to recover is not needed at this level, we want to make sure that the communication layer will survive broken connections. Currently, layers like PAMI [14] and uGNI [15] are able to withstand faults in this sense, and this capability will be reflected by the LRTS.

Moving upwards, the next layer provides a data-driven scheduler in user-space, which can work from a prioritized queue when needed. It also provides user-level light-weight threads. Such threads are used in Charm++ [1], Adaptive MPI [16], Sandia's Qthreads project [17] and HPX [18].

The next layer provides the key mechanism that makes our approach possible: it supports our execution model based on a large number of migratable entities (WUDUs: see §2.1). The layer provides naming and partitioning schemes for these entities, and supports scalable location services, as well as migratable threads. Relevant previous work includes Chare Arrays in Charm++, isomalloc [19, 20], Qthreads [17], ParalleX [18], and work on scalable communicators in MPI [21, 22].

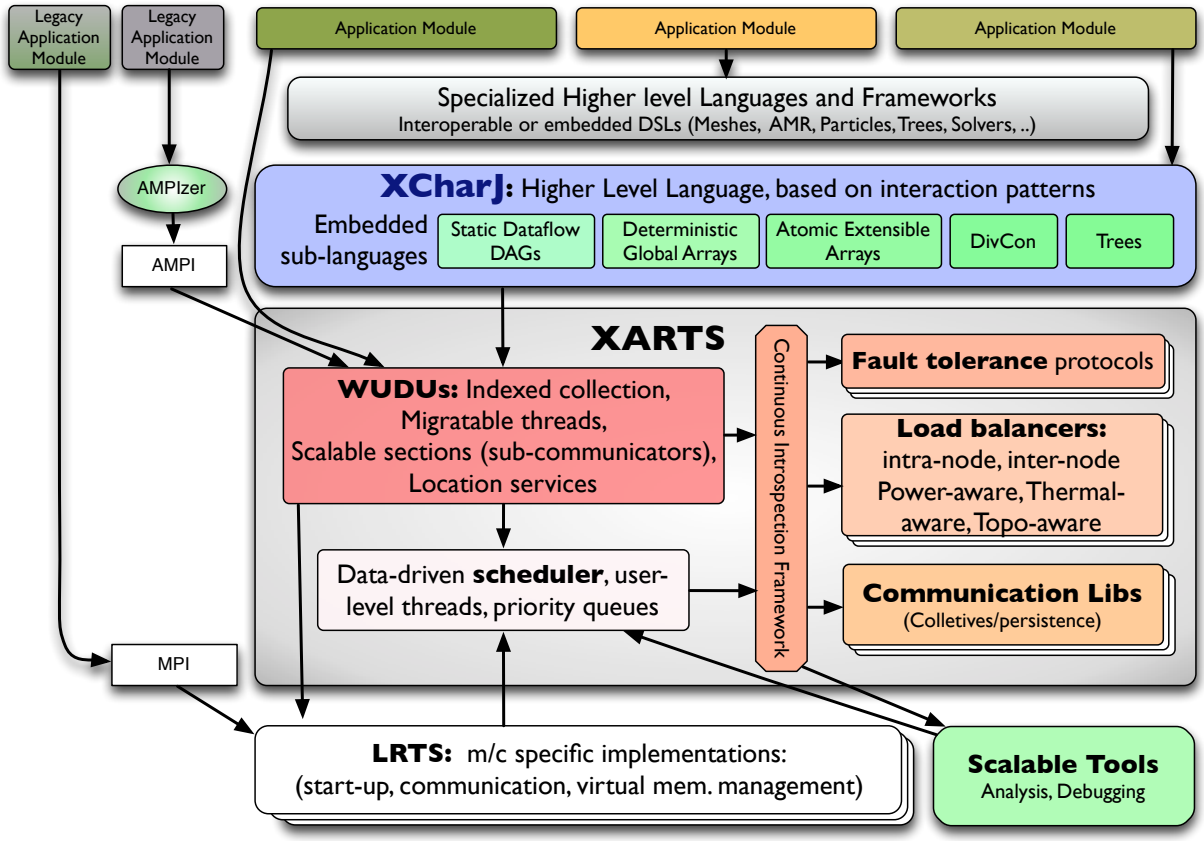


Figure 1: Proposed Work on the Exascale Stack in Perspective

We will develop a continuous monitoring framework (§3.3) to provide introspective capabilities XARTS needs to execute its adaptive strategies. There is relatively little existing work in this area, except for performance visualization [23–25].

A set of highly scalable adaptive strategies constitute a major portion of the proposed effort. Adaptive load balancers have been studied in projects such as Trilinos [26], Charm++ [1], and in more domain-specific ways in AMR frameworks such as Paramesh [27] and Chombo [28]. Our point of departure from many of them is that we will be deciding on migrating chunks (WUDUs) rather than the "elements" across processors, and we will do it in a generic application-domain-independent way, typically based on introspection.

Fault tolerance strategies have received significant attention in HPC recently [29, 30]. Our objectives here is to ensure low overhead, high scalability, and high rate of progress in presence of component failures. We will build upon several ideas in existing literature, and especially in ways that leverage the over-decomposition of our execution model.

Extensive research has been done on optimization and selection of communication algorithms as they are critical in obtaining good performance [31,32]. We propose a selection scheme that will use the observed persistent communication pattern, including the *context* of the communication operation, across iterations. Thus, if the pattern persists, it is equivalent to taking a decision in present while knowing the future.

Various tools, some of which have recently shown scalability to $200,000$ processors, exist for parallel debugging and analysis, including TotalView [33], DDT [34], Eclipse [35], TAU [36], Vampir [37], Jumpshot [38], Cray Apprentice [39] and HPCToolkits [40]. In our previous work, we have developed scalable techniques for debugging and analyzing message-driven parallel applications [41,42]. Using our execution

model, which allows integrating data collection with application communication, we will demonstrate the utility of our approach to implement scalable tools.

Recognizing that the abstraction level provided by MPI (or MPI+OpenMP) is inadequate, several research efforts have focused on developing new higher level languages, some of which are compiler supported. PGAS models support notions of globally shared data or arrays. Global Arrays [43] is a library based approach, whereas UPC [44] extends the C model of memory (including pointer arithmetic) to a distributed environment. CAF [4] provides a shmem like access to remote processors' data with a convenient notation, while its compiler optimizes it, e.g by doing aggregation. Languages developed by IBM (X10 [45]) and Cray (Chapel [46]) support PGAS, other asynchronous abstractions and compiler optimizations.

Our approach is different from the above mentioned: we think *interaction patterns* between parallel entities can be specialized to provide simpler, and in some cases, deterministic-by-design parallel languages, each of which is inadequate by itself for all parallel programs, but together provide a rich toolbox (§5). Domain-specific (and data-structure-specific) frameworks can be built on top of such a toolbox. Conscious support of interoperability, which requires careful design of features in the runtime, ensures that these tools form a synergistic ecosystem.

We begin by describing our execution model (§2.1). Our execution model, which is essentially same as that of Charm++ and HPX [47], is simultaneously a radical departure from the status quo represented by MPI (and MPI+ OpenMP), while having been tested on well established parallel applications (e.g. NAMD [48], OpenAtom [49], Cosmology [50]) for several years.

2 Foundational Concepts

We next describe the execution model and the tenets underlying our approach to exascale programming. Migratable, overdecomposed WUDUs form the basis of our execution model (§2.1), which empowers the adaptive runtime system (§2.3). We rely on the *principle of persistence* (§2.4), which is the basis of many adaptive strategies described later (§4).

2.1 Execution Model and Cost Model

The computation in our execution model consists of a large number of units. Each unit may be a work unit (WU), a data unit (DU), or more typically, an amalgam of the two (WUDU). An example of a pure work unit is a pure function that is given an input and produces an output without a side effect. A pure data unit is a slice of data that can be read or written as a unit from outside. Common examples of amalgams of the two include a thread with its own stack, or an object with encapsulated data and some methods to operate on. Since both extremes can be thought of as a special case of an amalgamated work unit/data unit, we refer to them all as WUDUs. In addition to the WUDUs, our ontology includes the notion of a trigger. A simple example of a trigger is an MPI-style message; but a trigger may also be a method invocation, a continuation, or a "kick" devoid of any data that is supposed to trigger some computation when it is received at a WUDU.

Each work unit is further broken down into dependent execution blocks (DEBs), each of which depends on one or more pieces of data that is not local to the work unit concerned. In addition, optionally, it may also depend on a condition local to the work unit, typically indicating the readiness of the work unit to undertake the computation of the DEB. A DEB's dependencies are satisfied when all its dependent data is locally available (though not necessarily in cache), and its conditions are satisfied. A DEB whose dependencies are satisfied is called a *ready* DEB. A DEB's execution is atomic, in the sense that it is not paused awaiting additional data (since there are no additional dependencies beyond the initial ones), nor is another DEB associated with the same WUDU allowed to execute before it completes.

Although not a part of the execution model, we also define the notion of a sequential execution block

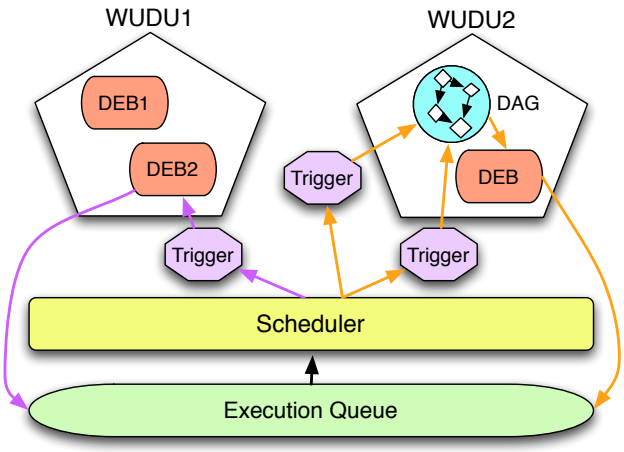


Figure 2: Execution within a processor

(SEB), for completeness. An SEB is a section of a DEB that is separated for analysis because of its cohesion. This could be a loop nest or a function call, or just an interval between two points in the execution of a DEB. Thus, a WUDU is a unit of migration, a DEB is a unit of scheduling, and an SEB is the smallest unit of analysis/introspection.

Execution proceeds by selecting an independent set of ready DEBs, and executing an arbitrary subset of them. Two DEBs of the same WUDU are considered dependent, and are not executed simultaneously. Execution of a DEB may lead to a change in the state of the WUDU that owns it, and to creation of new triggers, thereby adding to the set of ready DEBs. Execution continues in this fashion until one of the DEBs calls for termination of the program. The quiescent condition, when there are no ready DEBs, is not necessarily a ground for termination. It is possible to specify that a trigger be created when quiescence is attained, thereby starting the next phase of the computation.

An efficient way of ensuring *independence* and *atomicity* properties mentioned above is to anchor each WUDU to a single core, at a time. This is the default strategy we adopt. But we can relax it when needed, as long as these properties are upheld.

The Cost Model for the application developer, or for the higher level languages that translate to the level of WUDUs is as follows. Creation of each trigger has a cost depending on the size of the payload data on it, via a standard $\alpha + n\beta$ form, where α and β are expressed in (say) cycles. Accessing data within a WUDU is relatively cheap, whereas accessing data via triggers is expensive. Accessing data in other WUDUs directly is not allowed. (exceptions: §3.2).

This execution model is similar to that used in Charm++ for many years, as well as in models such as HPX. Its utility, which was relatively small at terascale for most applications, is much higher at exascale, because of its flexibility, and the increased need for an adaptive runtime system.

2.2 Message-driven Execution

The execution model is *message-driven*: since there are multiple WUDUs on each core, a user-level scheduler must schedule DEBs based on availability of a trigger. This confers performance and modularity benefits to the application: if one DEB is waiting for some remote data, another ready DEB is scheduled for execution. This generates an adaptive and *automatic* overlap of communication and computation. Further, it spreads communication over the iteration, unlike the current MPI compute-communicate paradigm, which leads to the network being used for only a small fraction of the time. As a result, one can avoid the need for aggressively engineered (and power-inefficient) networks.

Compositionality: Since executions of distinct modules can interleave in a message-driven manner on a single core, parallel composition of parallel modules is well-supported by this model [51].

2.3 WUDUs empower Runtime Systems

In our scheme, the programmer (at least at the base level) is still responsible for the grainsize decisions; but the grainsize decision can be made without reference to the total number of processors. For good performance the computation of a DEB should be large enough to amortize the cost of its trigger. Another consideration in deciding the grainsize is that of the memory overhead. However, none of these quantities depend on the number of processors. Further, application developers (or compilers) may produce parameterized code which makes it easy to change the grainsize (See §4.5).

As a consequence, this methodology produces an abundance of parallelism: the number of WUDUs (and dynamically, the number of ready DEBs) can be much larger than the number of processors. This *over-decomposition* fundamentally empowers the runtime system to optimize the execution in multiple ways (§4), without intervention from the application developer.

2.4 The Principle Of Persistence

At exascale, hardware will exhibit static heterogeneity (e.g. accelerators) as well as dynamic heterogeneity (e.g. failures and chip temperatures). In addition, applications will exhibit dynamically changing behavior. In this shifting landscape, our main friend is the *principle of persistence*, albeit only a partly reliable one. This heuristic principle states that, *in most CSE applications*, once the computation is expressed in terms of the natural objects (WUDUs), the computational loads of individual WUDUs and their communication patterns *tend to* persist over time. Note that this is only a heuristic principle, which arises from the fact that most CSE applications tend to be iterative in nature. Even when the situation changes dynamically, in most cases it changes slowly (as when atoms migrate across boxes in molecular dynamics simulations) or abruptly but infrequently (as in periodic dynamic refinements). Of course, there exist some applications for which the principle of persistence applies weakly. However, as a heuristic it is extremely useful for optimizing performance in a running program, for an adaptive runtime system. The analogy here is that of a cache hierarchy: although the *principle of locality* is only a heuristic, the microprocessor industry built several generations of architectural design around the idea of cache hierarchy based on this principle. Our stack will have complementary strategies that can be used when this principle does not apply. To utilize it when it does apply, we will develop the ability (§3.3) to keep track of application and machine behavior dynamically.

3 XARTS: Exascale Mechanisms

We next describe the capabilities we propose to build in our exascale adaptive runtime system (XARTS). As will become clear, it is an ambitious design. We will point out where we intend to build/extend on previously developed ideas, and where previous work can be used (mostly) as it is.

The base of XARTS will be low-level runtime system (LRTS), which provides a portable interface for the most essential communication functions. LRTS provides a portable API for basic communication and machine-specific functions, such as scalable job startup, efficient one-sided, two-sided and active-message style communication, and virtual memory management. We will use existing software, from the Unistack project or the existing LRTS of Charm++, with a few extensions, for the purpose of research proposed here. Similarly, we will reuse the data-driven scheduler, and instantiations of user-level threads that can migrate across nodes, from the Charm++ RTS.

3.1 Managing Collections of WUDUs

Since WUDUs are specified by the programmer without any reference to which processor they live on, XARTS must decide the location of each WUDU. Also, to deliver triggers to WUDUs, XARTS must keep track of the locations of WUDUs, in an efficient and scalable manner. The first component of XARTS supports basic implementation of WUDUs and scalable adaptive location services that help deliver data to them efficiently, in spite of migrations.

Planned Research: A naming and representation scheme that supports creation of new WUDU collections, and insertions of new WUDUs in existing collections will first be developed. The challenge here is a compact (say 64 bit) representation which is also efficient for translating between language-level names and their binary representations. For location services, a distributed hash table (or a distributed "directory"), with a selective caching of locations, with "home" processors that almost always know the location of an object, is effective (As described in [52]).

At exascale, we will need to extend these ideas for efficiency, utilizing (a) multiple replicas of the directory for reliability and efficiency and (b) leveraging the large number of cores on a node to provide a more responsive service. We also plan to research methods for proactively populating caches especially after load balancing, and good cache replacement mechanisms.

This layer also supports scalable creation and operations on "sections" (which are analogous to sub-communicators of MPI, but apply to collections of virtual WUDUs). Existing runtime systems, such as Charm++, provide an inspiration for this layer, but the proposed work goes beyond them in providing algorithms for these operations that scale to multi-billion WUDUs spread over a billion cores. Proposed work includes implementation of scalable distributed creation of sections, with multiple topology-aware adaptive spanning structures, and specialized collective operations. Our experience suggests that sections that only need support for reductions are created often. Formation of such unranked sections can be done with simpler algorithms, which we will develop. For ranked sections, one challenge is to support it without requiring storage proportional to the size of the section on any one processor, and especially not on *all* processors, which is the norm (admittedly sustainable on current machines, but barely so) for many MPI implementations today.

Efficient implementation of collectives over sections is another challenging problem. A good algorithm-selection strategy must take into account not just message size and number of processes but also the location of section members within the physical topology of the network and distance between adjacent members. Further adaptive optimizations are described later §4.4.

Since XARTS strategies will depend on migratability of WUDUs, various mechanisms are needed to make them migratable. For pure data units, and objects, this is straightforward. But for WUDUs that include a user-level thread, this is a major challenge. Current methods include a technique that exploits large virtual memory space to implicitly "reserve" (but not allocate) specific portions of address space on *each* node of the system [19, 53]. We expect this scheme to be challenged at exascale, in part due to relatively crowded virtual address space. Newer schemes, some compiler assisted, and others solely within the runtime will be developed for this purpose.

3.2 Intra-Node Parallelism

A node can be defined, for our purpose here, as the domain of a single OS image (process). Physical infrastructure defines a limit for the number of cores on a node, but one can split it into smaller domains for convenience. Current machine support up to 12-32 cores per node, and this number is likely to increase to several hundred at exascale. The prevalent programming method is to use raw pthreads or OpenMP within a node, and MPI across nodes. This method allows one to exploit shared memory (in contrast to putting an MPI process on every core), but complicates programming by requiring use of two different models.

Further, efficiency is compromised, at least in the current implementations, since OpenMP does not accord adequate importance to locality.

We believe that an object (WUDU) decomposition can utilize multicore nodes effectively, with a few extensions. First, the WUDUs enforce locality of reference. Note that each WUDU is allocated to only one *core* at a time, and a work-unit is allowed by default to access only its own data, except via explicit remote access constructs. Further, our base model allows access to data declared as readonly (which is replicated on every *node*). An extension of this idea is to allow passing pointers to memory owned by a single WUDU to others, but under very specific access contracts: a write-exclusive contract, where the sender promises not to read the data nor pass it to other WUDUs until the write is completed, and a read-only contract where the sender promises not to modify it until the read is completed. In a library setting, these contracts are harder (but not impossible) to enforce, but we also plan to provide language/compiler support for them (§5.1). Enforcing such contracts defines a semantic analysis of the use of the XCharJ programming model (specifically the library's API). Some dynamic analysis is possible and may be effective; but fundamentally a library cannot see how it is used within an application. However, the compiler can see both how the library is used, and especially important, the surrounding and even global context. This compiler support is an effective way of enforcing such contracts. More precisely, specialized analysis passes can be written to support such semantic analysis on the library API; such analysis passes can be either structural, or be defined using the data flow framework within ROSE. This approach supports raising the level of the API defined as a library API to that of a language API with full compiler support [54].

We also think there is scope for innovation in how memory is allocated and how data is laid out; e.g. a chunked allocation, rather than a linear allocation, might be beneficial, especially if accesses are linear. Another idea for chunking arrays was shown to be useful in [55] (which won a best paper award). This paper also demonstrates the performance benefits of our execution model: KD tree construction was shown to be on par or faster than a well-optimized versions based TBB.

Planned Research: We will develop and study application-triggered use-cases for within-node parallelism. We will then develop a set of mechanisms and APIs to cater to them; these will include extensions of the chunked-array abstraction, including multi-dimensional arrays (some concepts from HTA [56] will be relevant for this work). Layout innovations will include runtime changes in the layout, esp. for multi-dimensional arrays and arrays-of-structs, based on access patterns.

3.2.1 Mechanisms for handling Heterogeneity/Accelerators:

Several aspects of our programming model are advantageous in the presence of heterogeneous clusters and accelerators technologies. Recall that the application is broken down into a collection of WUDUs and DEBs that operate on the WUDUs (§2.1. Since the ready DEBs can be executed concurrently (i.e. no data dependencies between them), the runtime system is free to peek-ahead in the queue of DEBs, schedule the DEBs to accelerators (when possible), and preemptively move data as required before the DEBs execution (overlap of data movement with execution). This type of optimization could be advantageous for architectures such as the Cell processor, which was designed with a streaming execution model in mind. An initial effort to apply this type of programming model, specifically in the context of CHARM++, has already demonstrated that such an approach is feasible [57, 58]. Further, many DEBs will likely be executing the same code (i.e. same DEB executing on a different WUDU), which will allow for other optimizations, such as agglomeration and single instruction multiple data (SIMD) execution on a subset of the ready-to-execute DEBs, to take place. Such optimizations are applicable in the context of general purpose graphical processing units (GPGPUs), which use a single instruction multiple data (SIMD) execution model at the hardware level. We propose the development of such optimizations within the XARTS, which will allow a programmer to focus on expressing concurrency in their application using WUDUs and DEBs. The key aspect of the model is to express the concurrency in a flexible manner, so that execution of the application can conform to a variety

of hardware architectures, including accelerators and heterogeneous clusters. The XARTS will dynamically distribute, load balance, and otherwise manipulate the application across a heterogeneous set of cores based on runtime information. We also propose supporting yet to be released accelerator architectures, such as the many integrated core (MIC) design developed by Intel.

3.3 Continuous Introspection Framework

To exploit the benefits of the principle of persistence, we will develop a continuous introspection framework, which will efficiently maintain a database of recent behavior of the WUDUs and their individual code-blocks, using hardware counters, sensors and timers. Modern microprocessors provide extremely cheap (fast) access to such counters, which we plan to leverage; e.g. the cost of a timer call on the Sandy Bridge machine is 25 nanoseconds. It is feasible to maintain such a database effectively because all the events required to monitor the activity are accessible to XARTS. The placement of WUDUS on processors is done by XARTS, communication between WUDUs is mediated by XARTS and execution of individual DEBs on physical resources is also scheduled by XARTS. The WUDUs are relatively coarse-grained (as a guide to intuition, readers may assume tens of the WUDUs on a core; although in some some applications one may use thousands of them per core); so, the memory overhead of this database is modest on each core. Of course, at exascale it will not be feasible to copy this database on one processor, but that is an issue that the *strategies* will be able to deal with.

4 XARTS: Exascale Adaptive Strategies

The heart of XARTS, and a major thrust of the proposed work, will be a set of exascale adaptive strategies, which carry out various optimizations using mechanisms such as: migration of WUDUs, controlling frequency/voltage of chips, intercepting and optimizing communication etc. The strategies to be developed include: (a) scalable dynamic load balancing to handle load imbalances, whether application induced, hardware-induced, persistent or transient, (b) temperature-control to reduce cooling expense while protecting hardware, (c) power and energy management, (d) a suite of fault tolerance protocols, and (e) communication optimizers that learn at runtime. These will build upon our previous work on adaptive runtime, while overcoming the scaling limitations in them. We next describe the motivation, techniques, existing/preliminary work and proposed research on each of the topics.

4.1 Scalable Dynamic Load balancing

Our execution model encourages development of a two-level load-balancing scheme: the work is decomposed into WUDUs by the programmer, and the WUDU's are assigned to processors by the runtime system. For extreme situations, one may need schemes to split and merge WUDUs (See §4.5). But in general, the runtime system needs to concern itself with WUDUs. Even on a 100 million core system, one expects typically only a few billion WUDUs at the most, which is a manageable number for a parallel runtime system.

Charm++ has demonstrated the utility of such a two-level scheme on machines with tens of thousands to over one hundred thousand cores. Object computational loads and their communication is recorded on each core. On smaller machines (up to several thousand cores) this graph is collected on a single processor, and the selected load-balancing strategy can use this graph to produce a new assignment of objects to processors. Charm++ also provides *hierarchical load balancers* that partition processors into groups, and send only aggregate information to a root processor. The root decides the aggregate movement of work from groups to groups and informs the group managers of this decision. The group managers send some appropriate pieces

of work (WUDUs) to the other group managers, and then do a detailed, accurate load balancing over their own set of objects.

Planned Research: In this project, we propose work on the following tasks:

Scalability of load balancer: In our previous work, we have successfully developed a general centralized load balancing framework, which improves parallel program performance in many real applications. This centralized load balancing framework is not feasible for exascale because the time cost to make the load balancing decision will be too high and the database of computation/communication tracing will not fit into memory on one node. We plan to add refinement strategies and further improve the multiple-level hierarchy load balancing framework to overcome this limitation. In addition, we plan to extend existing centralized strategies that take advantage of state of art graph partitioners such as Metis [59] and Scotch [60], to make use of parallel scalable graph partitioners provided in Par-Metis [61], PT-Scotch and Zoltan [62].

New Strategies for irregular applications: Previous work in Charm++ and other systems has focused on partitioning objects or blocks with communication graphs onto processors. These strategies work well with the scientific applications, which have clear iterative phases, with each phase having similar communication/computation patterns. We expect many exascale applications to exhibit a multi-phase behavior. Each iteration in such applications might have multiple phases, where each phase has disparate computation and communication patterns. An example is multi-stepping simulation of gravity in cosmology [63]. These require new strategies for achieving good performance. Further, some applications (with small iteration times) are significantly affected by transient load imbalances. We will develop new techniques to handle load balancing in these applications more accurately. We will also work on a load balancing strategy which examines the dependency graph of application (w.r.t WUDUs) and load balances based on critical paths and WUDU dependencies.

Topology-aware Dynamic Load Balancers Interconnection topology has become important on very large scale machines. If interacting entities are placed on faraway processors, the communication occupies a larger fraction of the bandwidth, leading to increased contention in the network. This is in spite of wormhole routing which makes no-load latencies almost impervious to the number of hops. It has been demonstrated, by us and others, that topology-aware mapping of objects to processors significantly improves performance [64–66]

Charm++ currently provides APIs that are used by application writers to discover the topology of machines. We propose to augment the centralized as well as the hierarchical balancers with the information provided by the API. To this end, we seek to develop new strategies for topology aware mapping that are both scalable and conducive to network characteristic of exascale machines. We also propose implementation of relevant schemes available in literature [67, 68]. In addition, as mentioned above, we plan to extend existing load balancer with topology aware features of graph partitioners such as Metis [59] and Scotch [60].

Meta Load Balancer: Another key task is development of a *meta load balancer*. It has been observed that load balancing can be very effective if certain key decisions are taken in accordance to application behavior. Examples of such decisions are frequency at which load redistribution is done, which metric to focus on during load balancing- computation and/or communication, and whether to perform a topology aware load redistribution. Taking advantage of the database maintained by Charm++ RTS, we propose a meta load balancer which will relieve application writers of such key decision making related to load balancing. This balancer will be capable of analyzing application's run time statistics and making optimal decisions (or giving suggestions if so desired by application writers). We believe automation of these key decision making will do away with any inefficiency that is introduced by usage of static conditions that are currently used for making these decisions.

Heterogeneous Load Balancer: In the context of heterogeneous systems, which may or may not include accelerator technologies, the runtime system will need to account for differences in the various processing elements. Given a set of subtasks that need to be executed, some tasks may be well suited for some subset of the processing elements. In such cases, a load balancing strategy should map those tasks to the

most appropriate type of processing element. However, none of the processing elements should be idle, even if they are not the ideal choice for any of the current tasks. Simply leaving the non-ideal processing elements idle wastes some of the computational resources and may result in non-optimal performance.

We propose the development of load balancing strategies that consider imbalances (e.g. in peak flop rates) between processing elements when distributing (or redistributing) a heterogeneous set of tasks. Further, we plan to develop load balancing techniques which distribute homogeneous set of tasks across a heterogeneous set of cores, so that all of the compute resources are taken advantage of, even if a subset of those resources are not ideal for the currently executing computation.

4.2 Energy Efficiency

4.2.1 Adapting to Thermal Variations

It has been estimated that 40% of the total energy spent by data centers is spent on cooling [69–71]. For exascale, where power and energy are major constraints, this will pose a challenge. Is it possible to save cooling energy? The reason machine rooms are kept cool is to avoid overheating cores, which will cause them to "burn out". Hot spots will develop in machine rooms because of room dynamics (which can be avoided by liquid cooling to some extent) but also from process variations i.e. some chips will overheat while others are operating within their safe range.

Modern microprocessors have temperature sensors that can be accessed via software. They also support DVFS features that allow the software to change frequency and voltage with low overhead. So, the runtime system can monitor the temperature periodically, and slow down the chips that are getting hot and speed them up when they cool down. Unfortunately, in a tightly coupled parallel program, such a technique will tend to slow all processors down, because of dependencies.

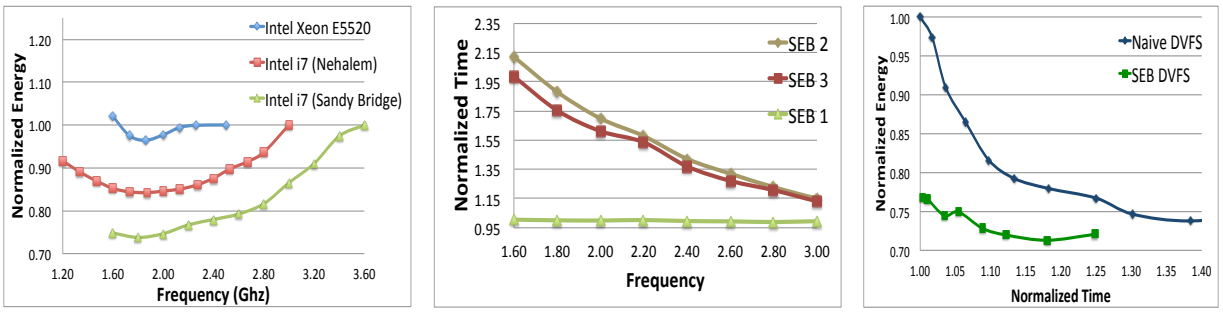
WUDUs help solve this problem: the runtime can migrate them away from the slowed down processors under the control of a more sophisticated load balancer. In fact, we demonstrated such potential in an experimental implementation using Charm++ [72] on a dedicated cluster where the cooling could be controlled. The scheme was able to demonstrate a 31% (on average) decrease in cooling *energy*. Without the load balancer, the cooling energy was reduced by 26 % (on average), because the program needed to run for a much longer time. However, the total energy, including the machine energy, increased by a considerable margin without load balancing, for the same reason.

This demonstration was done on a small cluster with 40 nodes. Also, the underlying application was chosen so it had no dynamic load imbalances. Further, exascale brings out new issues for temperature-aware load balancing, because of increased process variation in chips, and also because of the prospect of dark silicon. We plan to explore balancers that deal with application-induced load imbalances along with thermal variations. Utilizing turbo-mode (or its variants) also presents new challenges that must be addressed.

4.2.2 Minimizing Energy, Power, and Execution time

Leaving aside the issue of chips overheating, are there other ways to reduce energy usage in HPC applications at exascale? The fact that we can control frequencies, and the known cubic relationship of dynamic power to frequency has led to some research that exploits careful reduction in frequency as a way to reduce energy [73–75]. Of course, reducing frequency increases execution time, but the non-linear (cubic) reduction in power is expected to lead to overall energy savings.

However, the static power is a significant component of the overall power. This has led to folklore that "the best way to save energy is to run as fast as you can, and then switch off the machine". This was close to true on somewhat older (pre-Nehalem Intel processors, e.g.). In our experiments, even on Intel i7 (Nehalem) (see Figure 3(a)) the minimum energy is not at the lowest frequency. However, the trend is



(a) Energy on different architectures (b) SEB Execution Time (c) Time Energy Tradeoffs

Figure 3: Performance of NPB-FT

towards reducing static energy, and the most recent Intel i7 (Sandy Bridge) shows energy minima close to the lowest frequency.

Further, if we consider different sections of code (SEBs) separately, a more nuanced and interesting picture emerges. Figure 3(b) shows response of 3 different segments of NPB-FT to frequency change. We can see that some segment's execution time is impervious to frequency change, while others are highly affected by it. Thus, by selectively changing frequency/voltage for different SEBs, we can achieve a significant energy reduction, with little timing penalty (Figure 3(c)), in spite of the overhead of repeated frequency changes.

Planned Research: We first plan to develop mechanisms for runtime study of user-identified SEBs (code segments) to generate frequency-impact profiles for them. We will also develop schemes for automatically (perhaps with help from the compiler) demarcating SEB boundaries, so that each SEB is cohesive and homogeneous as far as power profile is concerned. We plan to develop strategies that utilize this ability of XARTS to monitor SEB behavior, and also the ability to balance load if needed, to provide a flexible tradeoff between energy, power and execution time. One can specify one (or two) of these as a constraint, and ask XARTS to minimize the other metric. We expect that future processors will provide increasingly flexible capabilities for this purpose. For example, instead of the current requirement to change the frequency for a chip as a whole, they may allow subgroups (tiles) of cores to be controlled separately (such as Intel's SC chip [76]). Such constraints make it a complex challenge for XARTS, for example by requiring coordinate scheduling of work on multiple cores. But certainly, XARTS is much better equipped to handle it, rather than the application programmer.

4.3 Fault Tolerance Schemes

It is expected that component failures will be more frequent in exascale computers [77], possibly one every few tens of minutes [78]. Soft errors, which do not halt a node, but cause results to be wrong, are also likely to increase. In the proposed work we will spend major efforts on dealing with failstop failures, and a smaller effort on soft errors. One reason for this emphasis is the scope of effort that is feasible within our project. The other reason is that a lot more can be done about failstop faults, and failstop errors will be a significant problem that must be dealt with. Many soft errors can be either contained locally (see §4.3.2) or be translated into failstop errors. Other work such as algorithmic fault tolerance [29] will complement our work.

Our execution model bestows two benefits for supporting fault tolerance. First, since it must support migration of WUDUs, it can use the same mechanisms to checkpoint a WUDU without requiring any additional programming from the user. Second, it is possible to restart or continue the computation on a different number of processors. We propose to work on two frontiers for dealing with failstop faults. The first, local

checkpointing, has adequate existing work to make it production-worthy within the first year. The second based on message-logging has the potential to tolerate a higher frequency of faults, while saving energy and maintaining higher progress rate, but requires a larger research effort.

4.3.1 Local or In-Memory Checkpointing

System level checkpointing to the file system is clearly impractical at exascale. Application-level checkpointing reduces memory footprint, but is still challenging at best from a performance standpoint. We explored, instead, the idea of checkpointing within the parallel machine itself, leveraging its communication network and storage capacity. The storage *can* be the local memory (DRAM) on nodes, for some applications, while for others one may utilize FLASH-style memories that are likely to be available locally on nodes of future machines.

The basic idea is to periodically create two checkpoints for each WUDU. One is stored in local storage (on the node) and the other is stored in the local storage of a buddy node. On failure, every node except the failed one restores their WUDUs from their local checkpoints, whereas the failed nodes WUDUs are restored (either on a spare processor or one of the existing processors) using checkpoints stored at a buddy. The two checkpoints constitute a memory overhead for this scheme, but (somewhat surprisingly) this is tolerable for a large class of applications: those that have a smaller memory footprint at checkpoint. These include molecular dynamics, N-body codes, certain quantum chemistry (namnomaterials code), etc. for others, the scheme relies on future local storage (or of course, a global file system as a fall back).

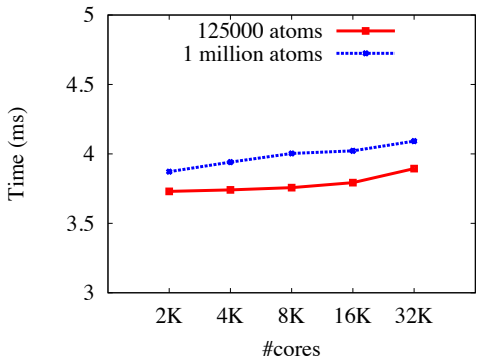


Figure 4: Checkpoint Time - Intrepid(leanMD)

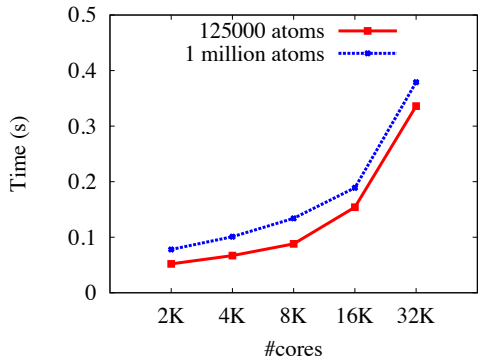


Figure 5: Restart Time - Intrepid(leanMD)

The idea has been explored in our previous work [79], and recently shown to be highly efficient. Figures 4 and 5 present the checkpoint time and restart time, respectively, for a molecular dynamics simulation [80].
Planned Research: Although the above results are very good, they are somewhat preliminary and can be further improved upon. We will incorporate the techniques into XARTS stack. The "pause" during checkpointing can be reduced by overlapping it with computation. However, the checkpointing (to buddy) traffic can interfere with normal computation in some applications. Also, faults occuring during such checkpointing are problematic (unlike the base scheme). We will develop new techniques based on interleaved streaming and selective duplication to eliminate these obstacles and reduce checkpoint overhead.

4.3.2 Message Logging with WUDUs

The motivation for this technique is to avoid sending all the nodes to their checkpoints when only one node has failed. It requires only the failed node to redo the work. But other nodes must resend it the messages they sent it since the last checkpoint, which requires all the messages to be logged, in addition to storing the checkpoints at the buddy node. Of course, since the failed node has to reexecute lost computation, it

takes roughly the same time as a checkpoint-based scheme to restart, except for the significant but second order effects created by the fact that the communication network is not congested. Also, energy use can be lowered during recovery. Further, computations that don't depend on the failed node may continue.

The main benefit of our model for this protocol is the potential for parallel restart: the WUDUs from the failed node can be scattered to other processors to recover in parallel. For example, we demonstrated recovery in 6 seconds for a job that failed 28 seconds after its last checkpoint. This technique allows applications to make progress in spite of a checkpoint period larger than MTTI.

Planned Research: The main challenge for this techniques is to reduce the time and memory overhead in fault-free execution. Also, it forms a complementary technique to in-memory checkpointing, and which is better depends on the ratio of memory in checkpointing vs messages, etc. We plan a sophisticated methodology that chooses the protocol on a per-WUDU basis to create an optimum protocol. Learning/auto-tuning techniques, and opportunistic compression (to trade off time vs storage) will also be pursued. By exploring a similar idea to send-determinism [81], we plan to use a language extension that minimizes the amount of additional information the protocol must keep, thus reducing the latency penalty. We will also explore opportunities to reduce the amount of memory required to store messages, particularly in the case of collective communication operations. Finally, we will develop compiler support for reducing the size of checkpoints, and team-based schemes that allow a flexible tradeoff between restart time and memory overhead.

4.4 Communication Optimizers

Logical communication operations are expressed at the level of WUDUs, but they are implemented at the level of nodes (i.e. it is the physical entities, the nodes, that send and receive data). This separation between logical and physical aspects can be exploited to optimize costs by interposing an introspection-driven layer between them.

For collective operations, such a layer can decide which algorithm to use for a given operation; e.g. a reduction could use a simple spanning tree algorithm or pipelined operation, or a reduce-scatter followed by gather. A selection based on certain parameters such as message size, system topology, system load, etc. provides substantial performance gains as shown in [31, 32, 82–85]. However, the optimal choice should depend not just on parameters, which are static or depends only on current state of system, as is done currently, but also on whether there are overlapping operations, and whether the operation is on a critical path. Alternative algorithms also provide a tradeoff between CPU overhead and communication latencies. Optimizing communication via such context-sensitive algorithm selection strategies, driven by the data collected by the introspection framework, will be the goal of our research.

Planned Research: Specifically, we will first implement multiple alternative communication algorithms for collective operations on sections. These algorithms will be designed and implemented with certain optimization goals such as minimum latency, maximum bandwidth, short message etc. We propose development and implementation of techniques to determine the criticality of a given communication operation, and relative importance of overhead at that point in the application. These techniques will use the information accrued over multiple iterations of the application to predict the above mentioned characteristics of an operation. Finally, we will develop heuristic schemes that can choose and switch among these algorithms for specific operation instances in the application based on the predictions done by our techniques.

4.5 Auto-tuning Based on Control Points

The picture we have so far is that of a runtime system that is subservient to the application: the application creates units of decomposition (WUDUs) and hands them over to XARTS to map. When scheduled, a work unit will create triggers (messages) that it hands over to XARTS to deliver using any of its strategies. But there is no way for XARTS to *reconfigure* the application itself. *Control Points* is an idea to allow such

reconfiguration. Think of a control point as a knob, with a description of its effects on standard observable performance metrics, such as grain-size, message-size, etc. The application (in concert with the compiler (§5)) provides such a knob to XARTS. Using its knowledge of the current runtime conditions, XARTS may ask some application components to reconfigure themselves, if they have provided relevant control points. A recent PhD thesis from the PI's group explored this idea in a few simple contexts, and demonstrated its benefits [86, 87].

A simple example is a pipelined communication scheme for a Gauss-Seidel like application. The code will be pipelined, but the width of the pipeline can be changed by the runtime by calling a control point created by the compiler. Similarly, the RTS may ask for grainsize adjustments. This facility allows for fine-tuning of application behavior based on the configuration of the hardware that it is currently running on, without hard-coding assumptions about that hardware into the application itself.

Planned Research: At runtime, the system can use the knobs in multiple ways. We plan to develop several strategies in XARTS to navigate the decision trees based on performance metrics observed by XARTS. This can be thought of as run-time auto-tuning. However, one difference from traditional auto-tuning is that the "control system" uses the knobs in a directed manner (rather than searching) based on desired effects. If XARTS detects that finer-grained parallelism will be useful (say for load balancing, and if the chunks are too coarse), it can look for the right category of knobs, and turn them in the known "right" direction. Another auto-tuning idea we plan to explore is to have different processors explore a different region of the parameter space to identify good configurations – when a search is considered desirable. Thus, we turn the power of the parallel machine to our advantage, by sacrificing performance over a few iterations for improved performance over the lifetime of the application.

4.6 Scalable Debugging and Analysis Tools

Our execution model presents new opportunities and challenges for scalable tools. Its message-driven execution is useful to collect data for live analysis/visualization. The ability of the runtime to instrument at the level of application entities (WUDUs), rather than at the processor-level data collected by current MPI-based tools, is also beneficial for performance tools that can provide feedback at the application level. However, the fact that execution of multiple modules can interleave on the same processor (overlapping even wait times of different modules) adds new wrinkles for analysis. Although, due to resource constraints, we will not devote a large share of our effort to scalable tools, we do plan to carry out a few specific demonstration projects to evaluate the utility of our ideas to scalable tools.

Planned Research: We will develop a scalable live performance data collection module (and demonstrate it with simple client side visualizations). The essential idea of leveraging message-driven execution for this is described in [88]. In addition to simple profiling, we will develop a few sophisticated distributed analysis modules for this purpose. We will also demonstrate the idea of "deathbed" analysis — instead of the usual post-mortem analysis, we can utilize the parallel machine to carry out some automated analysis at the end of a full-scale parallel run. Using the resources of millions of cores for less than 2-3% of execution time, one can indeed do much powerful analysis, but it needs automation of the analysis process. We will only demonstrate the possibilities in this project.

5 XCharJ: MultiParadigm Programming with compiler support

A basic programming system for using XARTS can be constructed simply by providing an API. By going one step further and providing some interface-description support, the system can support a richer variety of features, for example the automatic generation of serialization/deserialization code. This is the approach that Charm++ took, albeit with a somewhat simpler (terascale) RTS. Such a system raises the level of

abstraction in one sense, by automating labor-intensive programming tasks such as object mapping, dynamic load balancing, fault tolerance, and energy management.

However, despite the benefits of this increased abstraction, the "feel" of such a language is very similar to MPI: explicit communication among multiple control entities, with very fine-grained manual control of low-level application entities. To address the exascale programming challenges a higher level of abstraction, and a correspondingly higher-level language is required.

Currently there is widespread acceptance of the idea that our existing programming methodologies are insufficient for the challenge that exascale computing provides. Therefore, we believe that there is a relatively small window of opportunity over the next several years in which the community will be open to experiment with such new languages and new approaches that go beyond the well-established "C/C++/Fortran plus MPI" approach that is dominant today.

Specifically, we propose to develop (a) multiple separate notations and abstractions for the few distinct specialized *interaction patterns* that we believe are dominant in parallel applications, and (b) an accompanying embedding language that incorporates these special notations as well as a general-purpose base language to cover patterns not expressible in them. Within this framework, we will develop analysis techniques that optimize performance and support elegant syntax to help enhance productivity (and create beautiful, clear, and concise code).

There are several benefits to developing parallel applications in such a multi-model manner. It enables freer choice of libraries and modules and encourage code re-use. They allow a "right tool for the right job" approach. For example, an array-based model can be used for array-intensive parallel code while a model specialized for tree-based parallel computations can be used where trees are the central data structure.

The use of multiple programming models (interaction modes) can also allow for greater analyzability and more expressive syntax. For example, consider *static data flow*, where a fixed set of objects exchange messages with the same neighboring objects, iteration after iteration. This is a common pattern that covers many algorithms (and complete applications). Given the constrained nature of communication allowed by this interaction pattern, the compiler can analyze the code to perform optimizations such as inference of communication collectives, and setup of low-overhead, persistent communication channels between communicating entities. Therefore, if we could design a notation specialized for this interaction pattern, we would create a more elegant, analyzable sub-language, albeit one that cannot cover all applications.

This *incompleteness* can be addressed by two schemes: first, we will develop a suite of such specialized notations for covering different interaction patterns, building upon library-level past work [89, 90]; second, we will ensure that they interoperate with each other, and with a base language. The specific notations are described in the subsections below. The embeding language, called XCharJ, will have Java-like syntax for sequential behavior, and will be designed for analyzability. As outlined in § 5.6 we will also make all XCharJ features available in C++ via ROSE [5]. This will give application programmers the option of programming using the new syntax via the XCharJ translator (which will produce C++ to be analyzed via ROSE), or to use the C++ API, which will be lower-level, but in a familiar language.

5.1 Deterministic Access to Global Data

Shared memory applications are often plagued by concurrency bugs such as race conditions and deadlocks. These bugs may be extremely subtle and require significant time and effort to detect and eliminate. This makes models that can guarantee that such bugs will not occur very attractive. MSA was developed based on the observation that applications that use shared arrays often access them in distinct phases [90]. Within each phase, all accesses to the array use a single mode, in which data is read to accomplish a particular task, or updated to reflect the results of each thread's work. MSA formalizes this observation by requiring *synchronization points* between *phases*, and limiting the array to only be accessed using one of several specially defined *access modes* (for example, read only, exclusive write, and accumulate), during each phase.

MSA is implemented as a C++ library that is tightly integrated with the Charm runtime [91]. It makes clever use of the C++ type system and template functionality to staticly enforce some aspects of the programming model, relying on runtime checking for the remaining cases. However, the lack of special syntax in MSA makes it easy to mistake remote array operations for local array accesses, because they are syntactically identical. Moreover, MSA's mechanism for enforcing semantics through the C++ type system is inflexible and difficult to extend beyond the simple cases of read-only and write-only access. In addition, because there is no distinction in the MSA programming model between elements stored locally and elements which must be fetched from another node, simple array accesses are inefficient because it must first be verified that the element to be accessed is locally available. In particular, in code with a regular pattern of array accesses it is much more efficient to fetch all the elements to be accessed first, and then perform all the accesses without performing any checking. However, due to MSA's library implementation this approach is quite cumbersome and forgoes even the limited syntactic sugar that C++ operator overloading provides to protected array accesses. This lengthens and complicates application code unnecessarily and discourages high performance code in favor of a simpler but less efficient expression.

Planned Research: We will develop a notation inspired by MSA, but extending its concepts, and embed it within XCharJ. With this, we can take advantage of the opportunity to provide improved semantic checks at compile time while also improving syntax. There is also scope for beneficial optimizations that will allow programmers to write simple code while attaining the performance associated with much more complex code. Furthermore, we are free to pursue access patterns that would be difficult or impossible to enforce using the C++ typed handle approach. We could support array permutation operations and array subsectioning, thereby allowing the division of an array into distinct subsets, each of which follows its own series of access phases while remaining part of the same logical array. This array subsectioning would imply a corresponding machine partitioning, in order to control the cost of synchronization.

5.2 Global Extensible Arrays with Transactions and Atomicity

There are some situations and interactions between parallel entities that are not covered by the disciplined (or restricted) deterministic access model. Unstructured mesh refinement provides an example, and so do many graph algorithms. A global address space is often useful in describing the algorithms, but atomicity (rather than determinism) suffices for controlling concurrency.

Of course, because locality is still important, the unstructured mesh must be chunked into partitions, just as one does with MPI. But one should be able to address nodes and elements of the mesh using a global naming scheme. One should be able to atomically (or transactionally) read and modify individual node and element records. Also, one should be able to add and delete nodes/elements, e.g. to carry out refinements and coarsening in a mesh.

Planned Research: The mechanisms we mentioned above are useful for applications and interaction behind unstructured meshes (e.g. covering graph algorithms). We plan to develop (a) a notation that supports global "collections" or arrays, indexed by sparse indices, to allow dynamic insertion without significant coordination; (b) global name generation schemes, and schemes for conversion of global indices to local contiguous indices for efficiency; and (c) synchronization primitives, including software transactions and atomic blocks that work efficiently across nodes, using low-level synchronization (and shmem) primitives provided by the machines.

5.3 Static Data Flow

Many CSE applications are iterative, and moreover, exhibit the same communication pattern iteration after iteration. That is, the communication pattern is *static*. Of course, the content and the size of the communication may vary over iterations. This is especially true when applications are expressed in terms of their

logical units (WUDUs) rather than in terms of processors. In particular, an algorithm that involves several activities on each processor will require the posting of wild-card receives in MPI. The same algorithm can be expressed more simply in terms of the flow of data and transfer of control between individual objects, leading to a *data flow* style of programming.

The Charisma notation was designed to support this pattern of *static data flow* [89]. Each Charisma object is allowed to communicate with a static set of objects through the consumption and production of data. Programmers are given a global view of control flow, which is specified in the form of a high-level *orchestration* script, together with separately defined sequential methods. The ability to specify global control flow makes programs more elegant to write and easier to analyze for correctness. A compiler analyzes the sequence of orchestration statements and the data items produced and consumed by each to infer the control and data dependencies in the program. The compiler then generates Charm++ code that encapsulates the message-driven interactions between the objects. Research has shown that for applications where the mold of static data flow suffices, it is highly productive and concise for describing interactions.

```
foreach i in Stencil
  <left[i],right[i]> <= Stencil[i].publishBoundaries();
repeat
  foreach i in Stencil
    <+error,left[i],right[i]> <= Stencil[i].compute(left[i+1],right[i-1]);
until (error < threshold)
```

< consider if the above code fragment and possibly a picture, is useful here. >

Charisma also enhances productivity through guarantees of determinism [92]. Although Charm++ does not guarantee message order, the Charisma compiler ensures that generated code respects two rules: (1) the statement that produces a certain data item always completes before the statements that consume it, and (2) methods are executed on a single object in the order specified by the program text. Together with a few other restrictions on program structure, these rules ensure the deterministic evolution of object state.

A higher-level language that encapsulates Charisma-like ideas would allow the user to define transition points where local control flow would transition to global, and data would flow easily into and out of this global coordination code. This feature is necessary to ensure tight integration with MSA and other C++ libraries, as well as other programming paradigms (described shortly) and plain Charm++ code.

We also propose compiler-level support for scheduling optimizations, enabled by static data flow. Machine topology and heuristics for DAG scheduling (instead of relying on local, greedy scheduler priorities) can be applied when distributing WUDUs across the cores, and can be used to guide the scheduler in making better decisions. In addition, by integrating MSA, array pages can be partitioned (or migrated for the static section of the code) and distributed based on compile-time information obtained from the analysis of interaction patterns within Charisma code.

5.4 Divide-and-Conquer

The divide-and-conquer paradigm allows the expression of algorithms in a recursive manner: the input problem is formulated as a set of independent sub-problems, each of which can be solved by applying the same (recursive) procedure as was applied to the original input. Delaunay triangulation is one common example of this type of problem. Problems of this type are implicitly parallel – in principle each sub-problem may be solved by assigning it to a parallel sub-task. The interaction pattern here is that of one parallel task spawning a set of others, and possibly combining the results of each one.

We will provide a notation specifically tailored for expressing implicitly parallel programs in the functional divide-and-conquer paradigm. A program will be written as a set of abstract statements that produce

and consume data, thereby implying a DAG in which nodes are immutable functional work units and edges represent data dependencies. The runtime system will perform the mapping of work units to hardware resources. The functional nature of our notation will make it much simpler for the compiler to guarantee that the remapping of program entities is safe.

In addition, we will explore a suite of inter-related optimizations that can be performed by an intelligent runtime system. For instance, adaptive grainsize control can be employed to strike the right balance between parallelism and overhead by subdividing large problems and redistributing the resulting sub-problems, even in computations that yield nonuniform recursion graphs. Furthermore, hierarchical neighborhood schemes can be used to proactively balance load across large configurations of processors, while minimizing communication volume. Finally, we will devise a set-based notation to specify data-parallel operations in array-based divide-and-conquer programs, and implement a *delayed data movement* scheme to reduce communication costs for programs that exhibit generative recursion.

5.5 Tree-Based Algorithms

The tree data structure is vital to several fields of computational science. The ability of trees to segment data into spatially-related partitions finds use in applications as diverse as computational astrophysics, collision detection and data mining.

Tree-based algorithms share several features in common. Prime among these are (a) a distributed tree data structure and (b) a recursive procedure for the traversal of this data structure. We will adopt a two-step approach to provide programmers with reusable constructs that can be used to construct tree-based codes.

First, we will develop a framework that commonly needed functionalities of tree-based codes. It will comprise several modules, foremost among which will be a *chunked tree* data structure that provides seamless access to the nodes of a distributed tree. Chunked trees will capitalize on the spatial locality of tree-based codes to automatically coarsen the grain of remote data accesses. This data structure will be coupled with a software caching mechanism with write-back capability to (i) improve reuse of remotely fetched data and (ii) combine element updates at the source to reduce overhead of communication. The cache will allow sharing of data across objects on a processor and across processors in an SMP node. It will also provide an MSA-style phase-based access interface so that concurrent write-back updates may be made to the data without undue overhead. Next, an extensible traversal framework will be constructed, in which programmers can express different strategies for the traversal of the distributed chunked tree data structure.

With such a specialized framework in place, we will design a tree-based language for the manipulation of first class entities such as trees, nodes and particles (possibly with nonzero extent). The language will also support the recursive formulation of tree traversals. Given its focus on *structural* recursion, it will be distinct from the divide-and-conquer language of § 5.4, which will cater to the expression of *generative* recursion.

5.6 Compiler support through the ROSE framework

ROSE [8], developed at Lawrence Livermore National Laboratory (LLNL), is an open source compiler infrastructure that can be used to build source-to-source program transformation and analysis tools for large-scale Fortran 77/95/2003, C, C++, OpenMP, and UPC applications. Internally, it generates a uniform abstract syntax tree (AST) as its intermediate representation (IR) for input codes. Sophisticated compiler analysis, transformations and optimizations are developed on top of the AST and encapsulated as simple function calls, which can be readily leveraged by tool developers. ROSE is particularly well suited for building custom tools for static analysis, program optimization, arbitrary program transformation, domain-specific optimizations, performance analysis, and cyber-security.

We will construct a ROSE-based compiler infrastructure that performs several routine and relatively straightforward optimizations. These include automatic derivation of serialization code, detection of live

variables at the point of migration for load balancing or checkpointing, strength reduction from threads to a set of simple methods that don't require a user-level thread, and so on.

The XCharJ system will be integrated with the ROSE framework in the following manner. The XCharJ translator will produce C++ code as its output, with additional semantic information (in the form of pragma's or sub-types) to facilitate processing within ROSE's abstract syntax tree. Convenient syntax for the specialized notations will be incorporated into XCharJ. For example, regular array notations for MSA and sequential array operations.

The compiler infrastructure will create "control points" for application-guided reconfiguration based on observed behavior (§4.5). Use of complier support for transition is described later (§6.1).

5.7 Example: Molecular Dynamics

To illustrate a potential use case for the combination of different special-purpose notations in a single cohesive application, consider the case of long-range Particle Mesh Ewald (PME) calculations in a modular dynamics code.

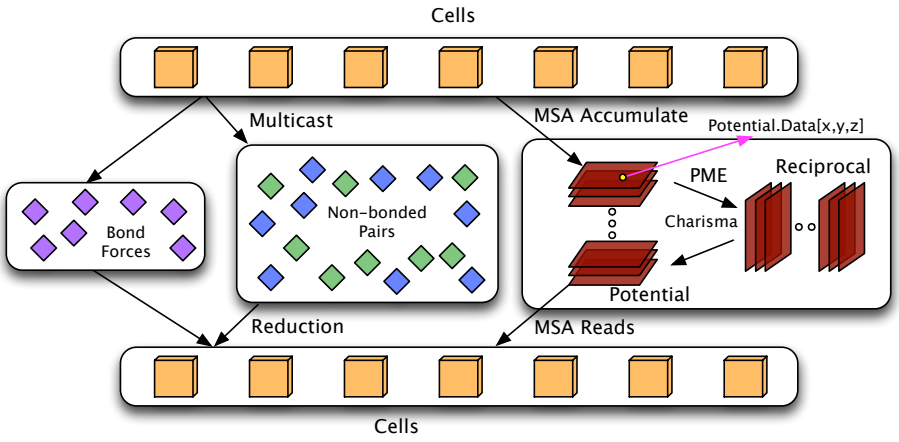


Figure 6: Program module interactions in a molecular dynamics application

In this application, the particles are divided among a collection of Cell objects, each responsible for the particles within a portion of the simulation space. The direct non-bonded forces among the particles are computed by Pair objects that receive particle positions from one or two Cells, calculate the interactions among the particles, and transmit the resulting forces back to the Cells. The Cells are responsible for integrating the net forces on the particles to obtain their velocity and position for the next time step.

For the PME portion of the computation, particles' charge contributions must be interpolated onto a *potential* grid. This grid is transformed into a reciprocal space via FFT. The long range forces are computed via convolution, then transformed back out of the reciprocal space and anterpolated back to the Cells.

Different phases of this process lend themselves naturally to different notations that we have described. The calculation of the potential using the charges of atoms in the Cells, (and its inverse at the end) can be done using complex point-to-point communication, but are more naturally expressed in terms of read and write operations on global arrays. These accesses have a natural phase structure, and so by using MSA we can achieve both a more convenient syntax and greater safety guarantees.

The FFT operation used to compute the long range forces and the communication between Cell and Pair Objects are natural examples of static dataflow. As such, they can be represented in Charisma in a concise and efficient manner. By adopting a global control perspective for this segment of the algorithm we can simplify the expression of the FFT while guarding against common bugs and maintaining high efficiency.

The overall algorithm can be described in XCharJ with a similar clarity as the figure (6 plus a manual explanation, but with a compiled language.

6 Transition Paths

XCharJ can be used to develop exascale applications from scratch. However, there is a significant amount of legacy MPI code within DOE and elsewhere. We plan to develop two methodologies to ease transition to our programming model, and to support interoperability with older code.

6.1 Converting MPI codes to use XARTS

We have developed Adaptive MPI (AMPI [93, 94]) that demonstrates that MPI programs can be converted with a small effort to use the adaptive runtime system. It implements each MPI "process" as a user-level *migratable* thread that is embedded in a Charm++ object. Threads are made migratable by using a virtual memory reservation trick: each thread allocates its stack in virtual memory in a space that is reserved for it on all processors. This reservation does not cost any memory: one mmaps a thread's stack in this space as the thread migrates to a processor. We have demonstrated that AMPI applications benefit from the load balancing provided by Charm++, e.g. in the Brazillian BRAMS weather simulation code [95, 96].

Planned Research: Although this scheme works well for developing new MPI applications, it requires avoiding use of global variables. For existing codes, this requires a conversion step, which can be an impediment to adoption of AMPI. In the proposed work, we plan to use ROSE to ease the process of converting MPI programs to AMPI by transforming applications to repackage their global variables and fixup all uses of such global variables. This automated step is a reasonable target as a source-to-source transformation using ROSE and has been the subject of both discussion and some initial work over the years.

Further, the scheme depends on a fragile scheme [19] of utilizing virtual memory reservations for migrating threads. We plan to develop new compiler support techniques, using ROSE, that lead to robust schemes for migratable user-level threads. This includes moving data from stack to the heap, and bounding stack size.

6.2 Interoperability with MPI

We will also support interoperability with modules written in plain MPI. This will be useful for modules that are not yet converted to AMPI (i.e. during the transition), as well as for modules that, for various reasons, will remain in MPI.

The challenge here is to join the disparate notions of control flow in XARTS/XCharJ and MPI. In MPI the control flow is explicit, and dictated by the program. In contrasts, XARTS message-driven scheduling causes interleaved scheduling of modules. The program only loosely controls the ordering; The scheduler is in control. We plan to meld these styles by means of a method for creating re-entrant XARTS scheduler, and supporting interoperability via time-sequencing, space-splitting and their combination as shown in Figure 7.
Planned Research: Normally, the scheduler in XARTS is deeply buried inside the "system". As soon as the system is initialized, it takes over control. We will create a new control structure that make the scheduler a "callable" entity. This requires separating initialization from scheduler, and creating data structures that persist between multiple scheduler invocations.

Library writers in XCharJ will provide a wrapper (possibly with assistance from the XCharJ compiler) that pure MPI applications can call. Essentially, the wrapper deposits the input data in XCharJ data structures, and calls the XARTS scheduler in a mode that allows normal return. When the library finishes, it signals to the scheduler to return control to the caller (the MPI program) on all processors where it was invoked. Multiple mechanisms for exchanging data between XCharJ and MPI modules will be investigated.

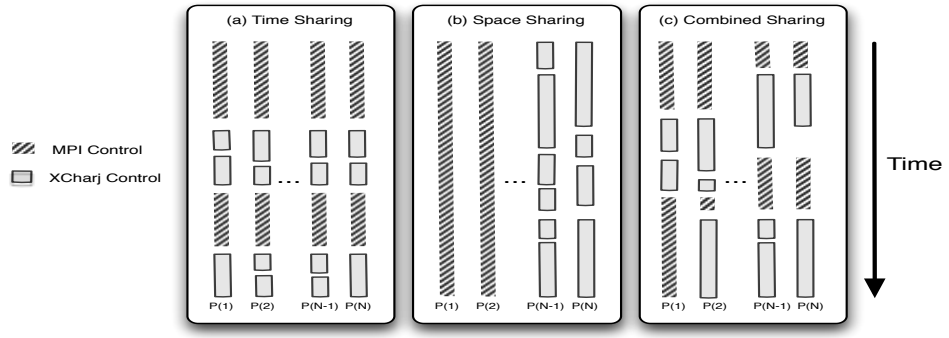


Figure 7: Different Modes of MPI-XCharJ Interoperability

Protocols for the reverse operation, where an XCharJ application wants to call a pure MPI module will also be explored.

Among the modes of control transfer, the first mode which we will support will be time-sequencing: control on all processors will transfer from MPI to XCharJ library on all processors, and return, like a collective operation, on all processors. The second, more complex, mode will be space-splitting. Here, a subset of processors will execute XCharJ module, while the main MPI program will continue on the the other processors. A more general version that allows space-time splitting and multiple entry-exits across MPI-XCharJ boundaries will be developed after that. Finally, we will incorporate techniques that allow backward flow: ability to call pure MPI modules from an XCharJ-based main application.

Interoperability with other models: The combination of message-driven and (MPI-like) explicit-control-transfer regimes is also expected to interoperate with other languages, such as X10 and Chapel. For example, the *Async* construct of X10 can be well-supported by our message-driven design. In addition, as our experiments with AMPI and Global Arrays [97] suggest, one can "virtualize" other models to fit them to XARTS. With Global Arrays, one can implement the "processes" as user-level threads, and over-decompose the data arrays to allow XARTS to optimize their mapping. We will cooperate with ongoing exascale language projects to support such interoperability.

7 Application Demonstrations

The DOE exascale co-design process and the International Exascale Software Program (IESP) have adopted skeleton, compact, and mini-apps as a central tool for driving co-design of computer architectures, programming models, and many other elements of the ecosystem. Since many of the relevant exascale design points center around data motion it is not sufficient to focus on computational kernels, though these are still interesting. Skeleton, compact, and mini-apps instead capture increasing levels of detail about communication and data motion patterns and volumes, ultimately including select full computational kernels. Each DOE co-design center will be producing multiple such entities with their collective efforts being organized by the co-design consortium (and RJH is a co-author of the original document founding the consortium). By shedding extraneous complicating details, mini-apps have also emerged as a good way of gauging the effectiveness of parallel programming models.

We will collaborate directly with the co-design centers and the co-design consortium to adopt their mini-apps (and those from Sandia's Mantevo collection) and also develop additional mini-apps representative of DOE application domains and algorithms not covered by the current three co-design centers (e.g., chemistry, nuclear physics, astrophysics, fusion, accelerators, etc.). Analysis early in the project of the mini-apps will drive design of the framework, and implementation in XCharJ of the mini-apps will guide refinement of the design, and ultimately enable demonstration of its capabilities and its evaluation.

The mini-apps we propose to develop are:

Barnes-Hut: a mini-app useful for testing many unusual use cases: fine grained access to global data, multiple interaction patterns during different phases of the computation, and complex load balancing requirements.
LeanCP: a simplified mini-app for Car-Parinello method, is useful because of its rich collection of diverse computational objects (WUDU collections), significant use of static data flow patterns, and need for optimizing voluminous partitioned-collectives communication.
LeanMD-PME: a molecular dynamics mini-app with Particle Mesh Ewald (PME) module included. LeanMD itself is identical to miniMD of the Mantevo suite. Adding PME for long range force evaluation tests the expressiveness in context of multiple modules.
Solid-fluid combined simulation, and
Multiresolution algorithms for nuclear physics and chemistry motivated by the MADNESS numerical environment.

In addition, LLNL will work with UIUC to evaluate the programming model within the context of a range of applications taken from the EXaCT Exascale Combustion Center. These application will be run at LLNL and support identifying and evaluating the XCharJ language and its compiler and runtime support. We will explore the appropriateness of XCharJ to support lab applications and those from EXaCT.

To further evaluate and demonstrate the capabilities of the framework and its interoperability with established programming models (MPI, OpenMP, Global Array, PGAS) we will incrementally transform elements of full applications into XCharJ. Currently being considered for this purpose include OpenAtom, NWChem, NAMD and MADNESS.

8 Management Plan

The overall project will be coordinated by the lead PI (Kale). Broadly, topics related to XARTS will be led by Kale, compilation/analysis by Quinlan in consulation with UIUC, applications by Harrison, with feedback/evaluation from LLNL applications and the combustion co-design center from Quinlan, and UIUC applications from Kale. XCharJ abstractions will be developed at UIUC with consulation and feedback from all the PIs.
Coordination Mechanisms: the PIs have participated in multiple collaborative development projects. Leveraging their experience, we will set up several on-line mechanisms for collaborations, including software repositories and wiki pages. We plan regular bi-weekly teleconferences, in person meetings every 6 months. Senior personnel at Illinois will track and coordinate milestones.
Evaluation Plan: We will evaluate the techniques, especially the WUDU management, and the adaptive runtime strategies on at least 250,000 cores. We expect to have access to such machines early in the performance period. Programmability will be studied using prevalent metrics such as lines of code (or unique tokens), as well as classroom evaluations by students. Large scale scientific study of programmability, although desirable, is not feasible within the proposed effort. The techniques will be evaluated in several mini-apps, and portions of full applications or frameworks that we are involved with, including NAMD, OpenAtom, ChaNGa, and MADNESS.
Prototype: of the the software modules embodying the techniques developed will be made available to the community via some form of open access.
Metrics for Success: Broadly, we will consider the research to be successful if the techniques developed are adopted by the CSE community. More direct metrics will be downloads of software modules developed. However, since the results of research take some time to make into production software, and additional time to be accepted, these may be difficult to demonstrate within the performance period; Yet, we do plan to measure these. Specific technical metrics for the myriad subtasks planned will be developed once the project commences.

Risk Analysis: Typical risks stemming from milestone management, or in changes in the environment, will be handled by regular review of progress and ongoing engagement with the community. We consider the following risks to be of the highest concern:

Risk: High Overhead of Over-Decomposition: This is a common misconception. Low overhead has been demonstrated for over-decomposed schemes, and by their use in challenging fine-grained applications (e.g. NAMD). That misperception will be mitigated by the high profile publication of compelling demonstrations and studies that carefully quantify the overhead. Situations in which overhead is a problem will be identified and addressed by specific techniques, such as message staging, and compiler optimizations to improve locality by "stitching" nearby regions together.

Risk: Lack of Community Acceptance of New Models. The PIs have long histories developing applications and in gaining acceptance of their ideas in the community. Continual evaluation in the context of mini-apps and complete applications, will ensure that the resulting software framework is efficient and useful for application development.

Section	Task	By	Lead
XARTS Mechanisms §3	Scalable Directory Service - Base Design / Optimization	Q4/Q6	UIUC
	Unranked sections creation and management	Q4	UIUC
	Ranked sections - collective operations	Q8	UIUC
	Compiler assisted migrations	Q6	UIUC
	Use cases for within node parallelism	Q10	UIUC
	Chunked array - 1D abstractions / multi-dimensional	Q7/Q10	UIUC
Introspective Framework §3.3	Survey of existing framework (Charm, HPX etc)	Q2	UIUC
	Framework design	Q5	UIUC
	Framework implementation	Q10	UIUC
Adaptive Load Balancing §4.1	Irregular Apps: multiphase, transient imbalance, dependency graph	Q3	UIUC
	Scalable LBs: hierarchical, distributed graph partitioning	Q5	UIUC
	Topology aware LBs	Q8	UIUC
	Heterogeneous LBs	Q9	UIUC
	Meta LB for automation	Q11	UIUC
Energy Efficiency §4.2	Mechanisms for SEB boundaries - manual and automated	Q4	UIUC
	Run time profiling of SEBs for impact of frequency	Q5	UIUC
	Tradeoffs between energy, power and time	Q7	UIUC
	Energy and Thermal balancers	Q9	UIUC
	Handling skewed applications	Q12	UIUC
Fault Tolerance §4.3	Implement semi-blocking cooperated check point restart	Q3	UIUC
	Utilizing local flash memory	Q5	UIUC
	Identifying limits of in-memory checkpointing	Q7	UIUC
	Optimization based on Teams	Q8	UIUC
	Selective scheme for in-memory vs message logging	Q9	UIUC
	Soft Error containment	Q10	UIUC
	Compiler support for reducing check points	Q11	UIUC
Communication Optimization §4.4	Implementation of section based operations	Q5	UIUC
	Methods to identify criticality and overhead of operations	Q7	UIUC
	Automated selection and switching among the algorithms based on runtime and persistent parameters	Q10	UIUC

Control Points §4.5	Decision tree navigation based on performance metrics	Q6	UIUC
	Cooperative configuration exploration	Q9	UIUC
Debugging and Analysis §4.6	Scalable live data collection module	Q6	UIUC
	Distributed analysis modules	Q9	UIUC
	Deathbed analysis	Q11	UIUC
XCharJ §5	Develop and implement syntactical constructs	Q1	UIUC
	Implement inter-model techniques	Q3	UIUC
	Integrate model-specific control points, FT and LDB	Q5	UIUC
	Generate straight C++ targeting XARTS	Q7	UIUC
	Generate ROSE-targeted pragmas in code output	Q10	UIUC
	Directly generate ROSE intermediate representation	Q11	UIUC
Deterministic Global Arrays §5.1	Design and notation development	Q2	UIUC
	Parsing and runtime implementation	Q4	UIUC
	Optimizations	Q9	UIUC
	Evaluation with applications	Q10	UIUC
Global Extensible Arrays §5.2	Design of the API	Q1	UIUC
	Extensible name generation	Q3	UIUC
	Global to local contiguous indices	Q7	UIUC
	Implementation and design of synchronization primitives	Q9	UIUC
Static Data Flow §5.3	Mechanisms for flowing data and control in and out of Charisma	Q3	UIUC
	Compiler-supported scheduling optimizations	Q5	UIUC
	DAG scheduling for Charisma	Q7	UIUC
	Evaluation using applications	Q11	UIUC
Compiler Support §5.6	Analysis and initial transformations for parts of XARTS	Q4	LLNL
	More complicated analysis to support XARTS and transformation to optimize the use of the runtime library	Q6	LLNL
	Define possible migration path for legacy codes to support semi-automated transformation to XCharJ	Q8	LLNL
	Development of semantic analysis to enforce semantics of XCharJ usage in large scale applications	Q10	LLNL
	Develop mixed static and dynamic analysis to support optimizations using high level XCharJ semantics	Q12	LLNL
Transition §6	Automate the refactoring of MPI codes to support AMPI	Q2	LLNL
	MPI-XCharJ Interoperability - time and space sharing	Q4	UIUC
	MPI-XCharJ Interoperability - combined sharing	Q8	UIUC

Application Studies and Feedback §7	Cooperation in design of abstractions	Q2	ORNL
	Analyze used target combustion applications for evaluation of XARTS and its compile-time support	Q4	LLNL
	Analysis of mini-apps from the point of view of adaptivity and abstractions	Q4	ORNL
	Guidance on prioritization of adaptive strategies needed for DOE applications	Q6	ORNL
	Assessment of overheads of the proposed techniques in the context of mini-apps	Q8	ORNL
	Retargeting MADNESS framework to XARTS	Q12	ORNL

References

[1] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[2] L. V. Kale and Attila Gursoy. Modularity, reuse and efficiency with message-driven libraries. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 738–743, San Francisco, California, USA, February 1995.

[3] Laxmikant V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[4] Yuri Dotsenko, Cristian Coarfa, and John Mellor-Crummey. A multi-platform co-array fortran compiler. In *Proceedings of the 13th International Conference of Parallel Architectures and Compilation Techniques (PACT 2004)*, Antibes Juan-les-Pins, France, October 2004.

[5] Daniel Quinlan, Qing Yi, Gary Kumfert, Thomas Epperly, and Tamara Dahlgren. Toward the automated generation of components from existing source code.

[6] L. V. Kale. Application oriented and computer science centered HPCC research. pages 98–105, 1994.

[7] Madness website. http://www.csm.ornl.gov/ccsg/html/projects/madness.html.

[8] Rose website. http://www.llnl.gov/CASC/rose/.

[9] Gasnet: A portable high-performance communication layer for global address-space languages, 2002.

[10] Jarek Nieplocha and Bryan Carpenter. Armci: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *J. Rolim et al. (eds.) Parallel and Distributed Processing, Springer Verlag LNCS 1586*, 1999.

[11] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. MPICH: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

[12] M.J. Koop, T. Jones, and D.K. Panda. Mvapich-aptus: Scalable high-performance multi-transport mpi over infiniband. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12, April 2008.

[13] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004.

[14] Pami website. https://wiki.alcf.anl.gov/old/index.php/PAMI.

[15] R. Alverson, D. Roweth, and L Kaplan. The gemini system interconnect. In *2010 IEEE 18th Annual Symposium on High Performance Interconnects (HOTI)*, 2011.

[16] Orion Lawlor, Milind Bhandarkar, and Laxmikant V. Kalé. Adaptive mpi. Technical Report 02-05, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 2002.

[17] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. Qthreads: An API for programming with millions of lightweight threads. In *International Parallel and Distributed Processing Symposium/International Parallel Processing Symposium*, pages 1–8, 2008.

[18] G.R. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu. ParalleX: A study of a new parallel computation model. In *EEE International Parallel and Distributed Processing Symposium, 2007*, pages 1 –6, march 2007.

[19] Gabriel Antoniu, Luc Bouge, and Raymond Namyst. An efficient and transparent thread migration scheme in the PM^2 runtime system. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) San Juan, Puerto Rico. Lecture Notes in Computer Science 1586*, pages 496–510. Springer-Verlag, April 1999.

[20] Isomalloc thread migration. //charm.cs.uiuc.edu/papers/migThreads.www/node18.html.

[21] Paul Sack and William Gropp. A scalable mpi comm split algorithm for exascale computing. In *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, EuroMPI'10, 2010.

[22] Humaira Kamal, Seyed M. Mirtaheri, and Alan Wagner. Scalability of communicators and groups in mpi. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 264–275. ACM, 2010.

[23] Jerry Yan, S. Sarukhai, and P.Mehra. Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit. *Software – Practice and Experience*, 25(4):429–461, April 1995.

[24] Brad Topol, John T. Stasko, and Vaidy Sunderam. PVaniM: a tool for visualization in network computing environments. *Concurrency: Practice and Experience*, 10(14):1197–1222, 1998.

[25] A Knupfer, H Brunst, and W E. Nagel. High performance event trace visualization. *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on*, pages 258–263, 2005.

[26] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[27] Peter MacNeice, Kevin M. Olson, Clark Mobarry, Rosalinda de Fainchtein, and Charles Packer. Paramesh: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126(3):330–354, 2000.

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

[29] George Bosilca, Remi Delmas, Jack Dongarra, and Julien Langou. Algorithm-based fault tolerance applied to high performance computing. *J. Parallel Distrib. Comput.*, 69(4):410–416, 2009.

[30] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cedile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vencent Neri, and Anton Selikhov. Toward a scalable fault tolerant MPI for volatile nodes. In *Proceedings of SC 2002*. IEEE, 2002.

[31] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications*, 19(1):49–66, Spring 2005.

[32] Jelena Pjesivac-Grbovic, George Bosilca, Graham E. Fagg, Thara Angskun, and Jack Dongarra. Decision trees and mpi collective algorithm selection problem. In *Euro-Par*, pages 107–117, 2007.

[33] TotalView Technologies. TotalView® debugger. http://www.totalviewtech.com/TotalView.

[34] Allinea. The distributed debugging tool (DDT). http://www.allinea.com/index.php?page=48.

[35] Gregory R. Watson and Craig E. Rasmussen. A strategy for addressing the needs of advanced scientific computing using eclipse as a parallel tools platform. Technical Report LA-UR-05-9114, Los Alamos National Laboratory, December 2005.

[36] B. Mohr, D. Brown, and A. Malony. TAU: A Portable Parallel Program Analysis Environment for pC++. In *Proceedings of the 3rd Joint Conference on Parallel Processing: CONPAR 94 - VAPP VI*, September 1994.

[37] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

[38] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with Jumpshot. *The International Journal of High Performance Computing Applications*, 13(3):277–288, Fall 1999.

[39] Luiz DeRose, Bill Homer, Dean Johnson, and Steve Kaufmann. Performance Tuning and Optimization with CrayPat and Cray Apprentice2. In *Cray User Group Meeting 2006*, Lugano, Switzerland, 2006.

[40] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: tools for performance analysis of optimized parallel programs http://hpctoolkit.org. *Concurr. Comput. : Pract. Exper.*, 22:685–701, April 2010.

[41] Filippo Gioachin. *Debugging Large Scale Applications with Virtualization*. PhD thesis, Dept. of Computer Science, University of Illinois, September 2010.

[42] Chee Wai Lee. *Techniques in Scalable and Effective Parallel Performance Analysis*. PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, December 2009.

[43] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *J. Supercomputing*, (10):197–220, 1996.

[44] Thomas Sterling Tarek El-Ghazawi, William Carlson and Katherine Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2005.

[45] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[46] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21:291–312, August 2007.

[47] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In *ICPPW '09: Proceedings of the 2009 International Conference on Parallel Processing Workshops*, pages 394–401, Washington, DC, USA, 2009. IEEE Computer Society.

[48] James Phillips, Gengbin Zheng, and Laxmikant V. Kalé. Namd: Biomolecular simulation on thousands of processors. In *Workshop: Scaling to New Heights*, Pittsburgh, PA, May 2002.

[49] Eric Bohm, Abhinav Bhatele, Laxmikant V. Kale, Mark E. Tuckerman, Sameer Kumar, John A. Gunnels, and Glenn J. Martyna. Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L. *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, 52(1/2):159–174, 2008.

[50] Filippo Gioachin, Amit Sharma, Sayantan Chakravorty, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn. Scalable cosmology simulations on parallel machines. In *VECPAR 2006, LNCS 4395, pp. 476-489*, 2007.

[51] Attila Gursoy and L.V. Kalé. Performance and Modularity Benefits of Message-Driven Execution. *Journal of Parallel and Distributed Computing*, 64:461–480, 2004.

[52] O. Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. In *Proceedings of ACM 2001 Java Grande/ISCOPE Conference*, pages 21–29, Stanford, CA, Jun 2001.

[53] Gengbin Zheng, Orion Sky Lawlor, and Laxmikant V. Kalé. Multiple flows of control in migratable parallel programs. In *2006 International Conference on Parallel Processing Workshops (ICPPW'06)*, pages 435–444, Columbus, Ohio, August 2006. IEEE Computer Society.

[54] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Proceedings of the 2nd conference on Domain-specific languages*, DSL '99, pages 39–52, New York, NY, USA, 1999. ACM.

[55] Pritish Jetley and Laxmikant V. Kale. Optimizations for message driven applications on multicore architectures. In *18th annual IEEE International Conference on High Performance Computing (HiPC 2011)*, December 2011.

[56] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, Maria J. Garzaran, David Padua, and Christoph von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[57] David M. Kunzman and Laxmikant V. Kalé. Towards a framework for abstracting accelerators in parallel applications: experience with cell. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[58] Laxmikant V. Kale, David M. Kunzman, and Lukasz Wesolowski. Accelerator Support in the Charm++ Parallel Programming Model. In Jakub Kurzak, David A. Bader, and Jack Dongarra, editors, *Scientific Computing with Multicore and Accelerators*, pages 393–412. CRC Press, Taylor & Francis Group, 2011.

[59] George Karypis and Vipin Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing*, 48:96–129 , 1998.

[60] Cdric Chevalier, Franois Pellegrini, Inria Futurs, and Universit Bordeaux I. Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multi-level framework. In *In Proceedings of Euro-Par 2006, LNCS 4128:243252*, pages 243–252, 2006.

[61] George Karypis and Vipin Kumar. A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In *Proc. of the 8th SIAM conference on Parallel Processing for Scientific Computing*, 1997.

[62] U.V. Catalyurek, E.G. Boman, K.D. Devine, D. Bozdag, R.T. Heaphy, and L.A. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*, pages 1–11. IEEE, 2007. Best Algorithms Paper Award.

[63] Filippo Gioachin, Pritish Jetley, Celso L. Mendes, Laxmikant V. Kale, and Thomas R. Quinn. Toward petascale cosmological simulations with changa. Technical Report 07-08, 2007.

[64] Abhinav Bhatelé, Laxmikant V. Kalé, and Sameer Kumar. Dynamic topology aware load balancing algorithms for molecular dynamics applications. In *23rd ACM International Conference on Supercomputing*, 2009.

[65] Tarun Agarwal, Amit Sharma, and Laxmikant V. Kalé. Topology-aware task mapping for reducing communication contention on large parallel machines. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2006*, April 2006.

[66] Hao Yu, I-Hsin Chung, and Jose Moreira. Topology mapping for Blue Gene/L supercomputer. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 116, New York, NY, USA, 2006. ACM.

[67] Abhinav Bhatele. Application specific topology aware mapping and load balancing for three dimensional torus topologies. Master's thesis, Dept. of Computer Science, University of Illinois, December 2007. http://charm.cs.uiuc.edu/papers/BhateleMSThesis07.shtml.

[68] Torsten Hoefler and Marc Snir. Generic topology mapping strategies for large-scale parallel architectures. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 75–84, New York, NY, USA, 2011. ACM.

[69] R. F. Sullivan. Alternating cold and hot aisles provides more reliable cooling for server farms. White Paper, Uptime Institute, 2000.

[70] R Sharma C. D. Patel, C. E. Bash. Smart cooling of datacenters. In *IPACK'03: The PacificRim/ASME International Electronics Packaging Technical Conference and Exhibition*.

[71] R. Sawyer. Calculating total power requirements for data centers. American Power Conversion, 2004.

[72] Osman Sarood and Laxmikant V. Kalé. A 'cool' load balancer for parallel applications. In *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*, Seattle, WA, November 2011.

[73] Barry Rountree, David K. Lowenthal, Shelby Funk, Vincent W. Freeh, Bronis R. de Supinski, and Martin Schulz. Bounding energy consumption in large-scale MPI programs. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 49:1–49:9, 2007.

[74] Vincent W. Freeh, David K. Lowenthal, Feng Pan, Nandini Kappiah, Rob Springer, Barry L. Rountree, and Mark E. Femal. Analyzing the energy-time trade-off in high-performance computing applications. *IEEE Trans. Parallel Distrib. Syst.*, 18:835–848, June 2007.

[75] Rong Ge, Xizhou Feng, Wuchun Feng, and Kirk W. Cameron. Cpu miser: A performance-directed, run-time system for power-aware clusters.

[76] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108 –109, February 2010.

[77] Franck Cappello. Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *IJHPCA*, 23(3):212–226, 2009.

[78] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. Exascale computing study: Technology challenges in achieving exascale systems, 2008.

[79] Gengbin Zheng, Lixia Shi, and Laxmikant V. Kalé. FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI. In *2004 IEEE International Conference on Cluster Computing*, pages 93–103, San Dieago, CA, September 2004.

[80] Laxmikant V. Kalé et.al. Charm++ for productivity and performance: A submission to the 2011 hpc class ii challenge. In *HPC Challenge, SC*, 2011.

[81] Amina Guermouche, Thomas Ropars, Elisabeth Brunet, Marc Snir, and Franck Cappello. Uncoordinated checkpointing without domino effect for send-deterministic mpi applications. In *IPDPS*, pages 989–1000, 2011.

[82] Ahmad Faraj, Sameer Kumar, Brian Smith, Amith Mamidala, John Gunnels, and Philip Heidelberger. Mpi collective communications on the blue gene/p supercomputer: algorithms and optimizations. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 489–490, 2009.

[83] E.W. Chan, M.F. Heimlich, A. Purkayastha, and R.A. van de Geijn. On optimizing collective communication. In *Cluster Computing, 2004 IEEE International Conference on*, pages 145 – 155, sept. 2004.

[84] Rolf Rabenseifner and Jesper Larsson Träff. More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems. In *PVM/MPI*, pages 36–46, 2004.

[85] T. Hoefler, J. Squyres, G. Fagg, G. Bosilca, W. Rehm, and A. Lumsdaine. A new approach to mpi collective communication implementations. In *Distributed and Parallel Systems - From Cluster to Grid Computing (DAPSYS'06)*, pages 45–54. Springer, Sep. 2006.

[86] Isaac Dooley. *Intelligent Runtime Tuning of Parallel Applications With Control Points*. PhD thesis, Dept. of Computer Science, University of Illinois, 2010. http://charm.cs.uiuc.edu/papers/DooleyPhDThesis10.shtml.

[87] Isaac Dooley and Laxmikant V. Kale. Control points for adaptive parallel performance tuning. November 2008.

[88] Isaac Dooley, Chee Wai Lee, and Laxmikant Kale. Continuous performance monitoring for large-scale parallel applications. In *16th annual IEEE International Conference on High Performance Computing (HiPC 2009)*, December 2009.

[89] Chao Huang and Laxmikant V. Kale. Charisma: Orchestrating migratable parallel objects. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 2007.

[90] Jayant DeSouza and Laxmikant V. Kalé. MSA: Multiphase specifically shared arrays. In *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, September 2004.

[91] Phil Miller, Aaron Becker, and Laxmikant Kal. Using shared arrays in message-driven parallel programs. *Parallel Computing*, 38(12):66 – 74, 2012.

[92] Pritish Jetley and Laxmikant V. Kalé. Static Macro Data Flow: Compiling Global Control into Local Control. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops 2010*, 2010.

[93] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, pages 306–322, College Station, Texas, October 2003.

[94] Chao Huang, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. Performance Evaluation of Adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[95] Eduardo R. Rodrigues, Philippe O. A. Navaux, Jairo Panetta, Celso L. Mendes, and Laxmikant V. Kale. Optimizing an MPI Weather Forecasting Model via Processor Virtualization. In *Proceedings of International Conference on High Performance Computing (HiPC)*, 2010.

[96] Eduardo R. Rodrigues, Philippe O. A. Navaux, Jairo Panetta, Alvaro Fazenda, Celso L. Mendes, and Laxmikant V. Kale. A Comparative Analysis of Load Balancing Algorithms Applied to a Weather Forecast Model. In *Proceedings of 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Itaipava, Brazil, 2010.

[97] Chao Huang, Chee Wai Lee, and Laxmikant V. Kalé. Support for adaptivity in armci using migratable objects. In *Proceedings of Workshop on Performance Optimization for High-Level Languages and Libraries*, Rhodes Island, Greece, April 2006.