Proc. *17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing* – PDP'09, 2009, pp. 119-126.

Phoenix: A Runtime Environment for High Performance Computing on Chip Multiprocessors

Avneesh Pant, Hassan Jafri, Volodymyr Kindratenko
National Center for Supercomputing Applications (NCSA)
University of Illinois at Urbana-Champaign (UIUC)
Urbana, IL, USA
e-mail: {apant|hjafri|kindr}@ncsa.uiuc.edu

***Abstract*—Execution of applications on upcoming high-performance computing (HPC) systems introduces a variety of new challenges and amplifies many existing ones. These systems will be composed of a large number of "fat" nodes, where each node consists of multiple processors on a chip with symmetric multithreading capabilities, interconnected via high-performance networks. Traditional system software for parallel computing considers these chip multiprocessors (CMPs) as arrays of symmetric multiprocessing cores, when in fact there are fundamental differences among them. Opportunities for optimization on CMPs are lost using this approach. We show that support for fine-grained parallelism coupled with an integrated approach for scheduling of compute and communication tasks is required for efficient execution on this architecture. We propose Phoenix, a runtime system designed specifically for execution on CMP architectures to address the challenges of performance and programmability for upcoming HPC systems. An implementation of message passing interface (MPI) atop Phoenix is presented. Micro-benchmarks and a production MPI application are used to highlight the benefits of our implementation vis-à-vis traditional MPI implementations on CMP architectures.**

Keywords-runtime; chip multiprocesors; MPI

I. INTRODUCTION

Chip multiprocessors (CMPs) consisting of multiple processing cores on a single chip have become commodity. Memory bandwidth and power consumption limitations, coupled with the continued march of Moore's law, are resulting in an ever increasing number of cores per chip [13]. As supercomputing enters the petascale era, we anticipate that upcoming systems will consist of a large number of fat nodes, consisting of multi-core processors with symmetric multithreading (SMT) capabilities, interconnected via high-performance networking fabrics [11,12].

Current state-of-the-art system software treats CMPs as arrays of symmetric multi-processing (SMP) cores, however there are significant fundamental operational differences between CMP and SMP architectures that need to be considered to achieve their true potential [10]. For instance, CMP processing elements tend to have smaller dedicated caches per core compared to SMP. The aggregate cache size of a CMP system can be up to an order of magnitude smaller than a similar sized SMP system. Applications on CMP will need to be threaded at a finer granularity level to reduce their working set size for efficient execution. The inter-core communication bandwidth and latency on a CMP can be an order of magnitude more efficient than communicating across cores on a SMP. It is expected that memory latency and bandwidth bottlenecks will be further exacerbated in CMPs, leading to poor application scaling. Furthermore, we expect there will be a decline in available network bandwidth per core due to core counts increasing at a faster rate than network bandwidth.

Execution of parallel applications on these architectures brings to light a number of new challenges while magnifying many old problems. For applications, there are challenges to be tackled in terms of programmability as well as an optimal runtime environment. Message passing paradigm, as used by message passing interface (MPI) [14], and partitioned global address space (PGAS) [15,18] paradigm, as used by UPC, CAF, etc., give the programmer flexibility in architecting the parallel application in that the programmer has extensive control over details of parallelization and the capacity to extract maximum performance from the underlying hardware resources. This flexibility, however, can be cumbersome to manage on CMPs due to their inherent complexity and scale. A sophisticated parallel runtime system can help deal with this complexity by efficient management of hardware resources while providing interfaces and semantics for better programmability.

Parallel programming paradigms that rely on current runtime systems are also not adequately tailored for execution on CMPs. MPI implementations, for example, treat CMP as a collection of SMP cores, thus losing the opportunity to exploit features that enable fine-grained parallelism. MPI tasks, implemented as heavyweight processes, incur high context switch overhead during scheduling. Additionally, communication between tasks on a node incurs unnecessary overhead due to additional memory copies across process address space boundaries. The operating system scheduler is responsible for scheduling of MPI tasks on physical processors. Configurable application specific scheduling policies are challenging to implement in this environment as they will require kernel-level modifications.

Software stacks for CMPs will be required to be cognizant of these issues. An integrated approach for scheduling of constrained resources to minimize contention

and oversubscription is imperative. We propose Phoenix, a modular and extensible runtime for parallel application developers, compiler writers and system researchers to use as a vehicle for researching parallel scientific computing on CMP platforms and as a fully optimized runtime framework on such systems. Phoenix runtime consists of a collection of interfaces for thread management, scheduling, locking, memory allocation, etc., explicitly designed to support the fine-grained parallelism present on CMP architectures. The layering of existing popular parallel programming paradigms, such as MPI, atop Phoenix will enable efficient execution of legacy applications on CMP architectures.

Our prototype implementation of MPI represents MPI tasks as lightweight Phoenix threads. A significantly larger number of MPI tasks than processors on a node can be spawned. This enables support for fine-grained parallelism by ensuring that the working set of a task can fit within the limited cache on CMPs. Context switch overhead is further reduced over conventional MPI implementations due to our use of lightweight thread abstractions to represent tasks. Since MPI tasks exist within the same address space, this approach leads to efficient communication between tasks within a node. The Phoenix runtime system is used to provide integrated scheduling of both MPI tasks and network progress threads. We show with micro-benchmarks and a production parallel MPI application that significant performance gains can be obtained over conventional MPI implementations with careful design of a system tuned for CMP architectures.

The rest of this paper is organized as follows. Section II gives a description of Phoenix, our intelligent-adaptive runtime system and its constituent components. Section III covers the MPI paradigm and its implementation on the Phoenix virtualization runtime. Section IV provides some preliminary results using micro-benchmarks and a production MPI code called MILC [1,2]. Section V covers related work and section VI concludes this paper.

II. PHOENIX RUNTIME ARCHITECTURE

The Phoenix core runtime (PCR) system provides a rich feature set for efficient layering of parallel programming models. The core functionality for scheduling, thread management, memory allocation, debugging, etc., is contained within the runtime system. The PCR provides a compact and expressive interface that is suitable as a target for a compiler backend and experimenting with new languages for parallel computing designed to scale from the desktop to large-scale systems. Each component of Phoenix provides a well-defined interface, enabling system designers to experiment with different policies and implementations of a given component by plugging them in at a defined application programming interface (API) layer. This flexibility is vital to explore a large design space for optimizations on CMP architectures [16].

Phoenix is designed to enable applications to scale from a single core to potentially all cores available on a node. In order to support layering of parallel programming models that span nodes, Phoenix provides a high-performance modular network layer with support for active message and remote direct memory access (RDMA) communication semantics. Fig. 1 provides the schematic view of the Phoenix ecosystem. This section provides an in-depth view of the Phoenix ecosystem.

A. Execution Contexts

Execution contexts (ECs) are the basic unit of execution in Phoenix. ECs are analogous to threads, and the Phoenix EC component API is representative of such. The API consists of routines related to EC management (create, exit, id) and synchronization (suspend and wait). The EC interface is implemented using POSIX threads. In order to support fine-grained parallelism present on CMP architectures, a large number of ECs need to be instantiated. Lightweight thread abstractions that can lead to a substantial reduction in memory footprint and context switch overhead are being investigated.

An important feature of Phoenix ECs is attribute tagging. Applications tend to execute in distinct phases. In each phase the resource requirements of an application can vary significantly. HPC applications, for example, during a given iteration, may compute for a sizeable amount of time followed by exchanging data with neighboring processors over the network. This pattern is repeated during the course of a run. Phoenix, therefore, features the notion of EC attributes for identifying the dominant characteristic of EC execution. The current supported attributes are: compute, network and I/O. An EC can change its attribute dynamically during execution of each phase. Attribute tagging will allow the exploration of various resource-aware scheduling strategies that will minimize contention of resources.

B. Processor Virtualization

Each schedulable processing element is represented by a virtual processor (VP) within the Phoenix runtime system. The VP API exposes primitives related to binding and

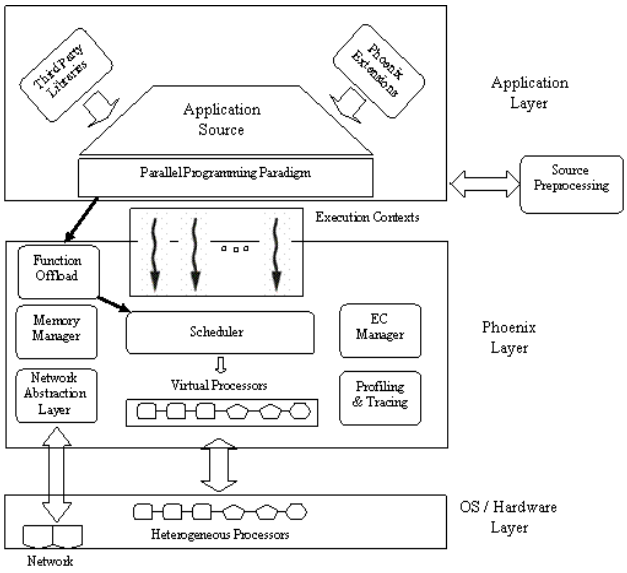

Figure 1. Phoenix ecosystem.

scheduling of ECs onto the physical processing element as well as functions to query the underlying physical characteristics of the hardware. In order to support identification of performance bottlenecks, the processor virtualization component keeps track of vital profiling information during the course of execution. Each VP maintains processor utilization, total context switches, idle and compute times for itself. This information can be queried during a run by end user tools or even Phoenix components such as the scheduler to make informed decisions

C. Scheduler

The Phoenix scheduler dispatches ECs for execution on VPs. Phoenix implements cooperative scheduling semantics. Higher level programming models utilize this to present the illusion of independent progress for all ECs. Pre-emptive scheduling of ECs on CMP architectures is not desirable as parallel applications, which exhibit distinct communication and compute phases, would benefit if entire cores were dedicated to their ECs during the compute phase. The scheduler provides a pluggable framework for Phoenix VPs, allowing implementation of processor agnostic scheduling strategies. Scheduling across processors is accomplished utilizing *scheduling domains*. A scheduling domain represents a group of VPs that exhibit common physical and performance characteristics. VPs dispatch ECs scheduled on a domain for the underlying processor.

In order to have effective control of scheduling policies, it is necessary to bypass the OS scheduler. The current implementation on Linux utilizes the native POSIX thread library (NPTL). NPTL is a 1:1 thread library in that threads created by the user are in one-to-one correspondence with schedulable entities in the kernel. The Linux OS scheduler is negated by ensuring that at most one EC—i.e., pthread—is in a runnable state per physical processor. This is accomplished by utilizing condition variables and semaphores respectively. Each EC waits on a semaphore, which results in the EC being suspended within the kernel. Phoenix schedules an EC by signaling the semaphore corresponding to the EC, resulting in the pthread associated with the EC transitioning to a run state within the kernel. This approach, coupled with the ability to bind ECs to processors, provides full control over placement and scheduling of ECs within Phoenix.

D. Memory Management, Instrumentation and Profiling

The PCR contains a memory management subsystem where custom allocation strategies can be implemented. Both AMD's Hypertransport and Intel's Quickpath interconnect are tending toward a non-uniform memory access (NUMA) model. The memory allocation and the runtime both need to be NUMA aware for efficient execution on these architectures such that memory regions are allocated local to the node the EC is executing on. Thread migration is another feature that can benefit from custom allocation strategies. As applications become dynamic in nature, load-imbalances across processors will become more pervasive. A common solution is thread/work migration to achieve load balance [9,17] where an isomalloc memory allocation strategy can be employed that allocates memory to each work unit/thread from mutually exclusive virtual memory regions. These examples point to the need for a modular memory management component to experiment with various memory allocation strategies.

The ability to capture performance metric data and profiling various blocks of code is critical for identification and resolution of performance bottlenecks. Fine-grained profiling available through the profiling layer allows for capturing relevant data regarding memory allocation/ deallocation for any given set of ECs to aid in postmortem analysis and debugging.

E. High-Performance Networking

The network component is highly threaded and completely integrated into the runtime. Independent communication progress with maximal overlap of computation and communication is achieved by having network ECs dedicated for communication. The scheduler is configured to either cooperatively schedule all EC types (compute, network and I/O) across available VPs or to have one or more VPs dedicated for communication progress. This provides the flexibility to allocate the required mix of resources for network- or compute-bound applications.

Phoenix provides a modular networking component to enable implementation of parallel programming models, such as MPI, that span nodes. The network component exposes an active messages (AM) API as well as native support for RDMA transfers. These communication semantics enable layering of the send/receive model of communication prevalent in MPI as well as global shared memory models utilized by PGAS languages such as UPC [18] and CAF [15]. The current prototype supports Infiniband and TCP with work underway to support iWarp [19] for RDMA-based Ethernet network.

III. Message Passing Interface on Phoenix

MPI is the *de facto* programming model for parallel computing on distributed shared memory machines. We have developed an MPICH-based implementation of MPI over Phoenix, called MPICH-Phoenix. This is a proof-of-concept implementation to show various advantages that Phoenix can bring to production-class parallel applications on CMP platforms.

By virtue of its design optimized for CMP, MPICH-Phoenix more effectively exploits CMP resources compared to mainstream MPI implementations [6,7], resulting in improved performance. The improvement in performance results primarily from high-performance shared memory communication, cache blocking and communication/ computation overlap.

Before we explain these factors at length, we will briefly describe a few important architectural and implementation details of MPICH-Phoenix. As mentioned earlier, each MPI task is mapped to a single EC in our implementation. Multiple MPI tasks may be active within a single process address space. This necessitates privatization of global variables to resolve collision of symbols among tasks executing within a single Phoenix process. Currently, we use

Elsa [3], a source-to-source translation tool for transforming C/C++ applications. The standard MPI compiler scripts encapsulate the privatization process, making it completely transparent to the user.

The existence of multiple MPI tasks within a single address space allows for true zero copy communication, whereby an MPI sender can directly deposit a message to the receiver's buffer without the intermediary copies customary in mainstream MPIs. Moreover, the integrated scheduling and mapping of ECs available through Phoenix allows for collocation of tasks on the same or nearby physical sockets/cores. Section IV shows the bandwidth curve of MPICH-Phoenix compared to other MPI implementations to show this advantage.

Fine-grained parallelism and threading is another important feature of Phoenix. This allows us to spawn many more MPI tasks than the number of available processors with minimal overhead. This is commonly referred to as *virtualization*. The ratio of MPI tasks to processors is called the *virtualization factor*. Virtualized execution environments have been shown to provide several benefits [8]. A virtualized MPI implementation enables us to implement a data-driven model of execution. In this model, an MPI task has dedicated use of a processor until a communication operation invoked by it cannot be satisfied immediately. The runtime, using the specified scheduling strategy, selects an MPI task in a runnable state for execution on the blocking processor. This allows for automatic overlap of computation and communication, resulting in efficient utilization of CPU resources. Once the offending communication operation has completed, the original task is made runnable and available for scheduling.

Virtualization also allows cache blocking in applications. Cache blocking is an often used strategy that exploits the memory hierarchy for higher performance. Cache-blocked algorithms attempt to increase cache hit ratios by improving data locality. MPICH-Phoenix provides free weak scaling to an MPI application as the virtualization factor is increased. Once a virtualization factor reaches a certain threshold, the working set would be small enough to fit inside the cache. If the task iterates a large number of times over the working set, the compute efficiency of the processor would climb due to better cache utilization. Results presented in Section IV highlight this in greater detail for the MILC QCD application. The free cache blocking, however, depends on memory access patterns of the application. We have seen no improvement in applications using algorithms already optimized for cache blocking, as in the LINPACK benchmark. Moreover, we believe algorithm such as FFT that are inherently unfriendly to hierarchical memory might not benefit from cache blocking.

Subsets of collectives commonly used in various scientific codes and typically found to be major obstacles for scaling of applications have been implemented. A streamlined implementation of these collectives is especially important for a virtualized environment due to the large number of virtualized MPI tasks. Optimized implementations for allreduce, barrier and broadcast, which exploit the two-level logical topology of MPICH-Phoenix jobs, are available. Optimized scheduling strategies for collectives that take into account the message transfer dependencies are being investigated.

IV. RESULTS

Our experimental testbed consists of the Abe [12] and QP [24] clusters at NCSA. Nodes on Abe consist of dual quad-core (8 processing elements) EM64T processors with a 4 MB L2 cache shared between each pair of cores. The QP cluster consists of dual dual-core (4 processing elements) Opteron processors with a 1 MB L2 cache per processor. A single data rate (SDR) Infiniband network is employed between nodes. Intel C and C++ compilers were used for all tests.

A. *MPICH-Phoenix Performance Analysis versus Selected MPI Implementations*

In this subsection, we show the impact of context switch overhead, optimized intra-node communication and integrated scheduling using micro-benchmarks and the MILC QCD application. The performance of MPICH-Phoenix is compared against OpenMPI-1.2.1 [7], MVAPICH2-1.0.2p1 and adaptive MPI (AMPI) [9]. The former two are well known open-source MPI implementations typically used on production systems. AMPI, built on the Charm++ [8] framework, makes for an interesting candidate for performance comparison since, like MPICH-Phoenix, it is also designed to execute MPI applications using processor oversubscription.

Fig. 2 shows the intra-node bandwidth for MPI tasks across all implementations on Abe. MPICH-Phoenix significantly outperforms the other implementations for large message sizes. OpenMPI and MVAPICH2 both utilize shared memory segments for communication within a node. This requires two memory copies between the source and destination process. Since MPICH-Phoenix tasks execute within the same address space, data is copied directly from the source to the destination buffer. Additionally, the Phoenix scheduler is able to co-schedule tasks on the same die to utilize the increased core-core bandwidth present on the chip. The peak bandwidth of 10 GB/sec is due to these tasks being co-scheduled on the same die sharing the same L2 cache.

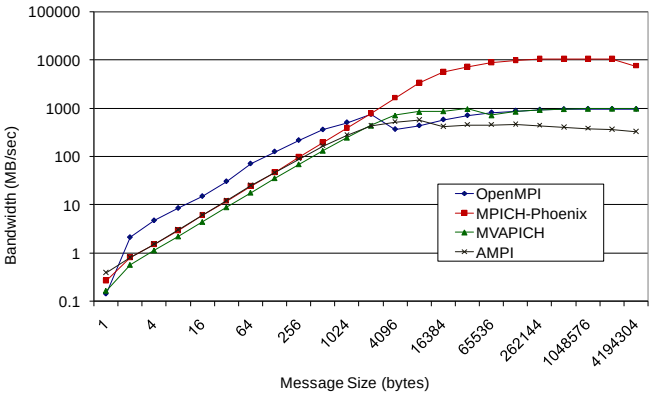

Figure 2. Shared memory bandwidth comparisong.

A common concern in the processor oversubscription model is the overhead of context switches. Standard MPIs typically create one MPI process per processor. However, OpenMPI supports a degraded mode of execution where processing resources are oversubscribed. In this mode, the MPI library yields the processor to allow its peers to make progress. This mode is similar to MPICH-Phoenix oversubscription with the exception of requiring the operating system scheduler for scheduling of tasks. The remaining comparisons with OpenMPI were done using this degraded mode.

To gauge the context overhead, a micro-benchmark was devised with each MPI task sending a small message to task 0 and executing a barrier. The benchmark is executed using a single core on a node with increasing virtualization factors for the core. Each MPI task, therefore, after sending a message, is context switched to allow the next task to run, allowing us to determine the context switch overhead. Fig. 3 shows the virtualization factor, i.e., number of tasks per core, and the ratio of context switching overhead using OpenMPI to MPICH-Phoenix. MPICH-Phoenix, where tasks are implemented as threads, coupled with lightweight user level thread scheduling, significantly outperforms OpenMPI.

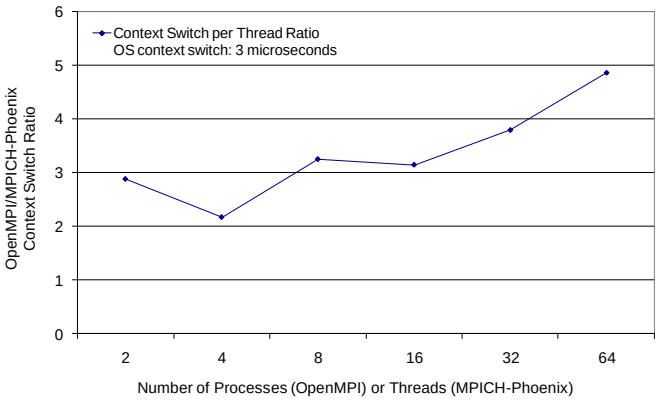

Figure 3. Context switch overhead of OpenMPI vs MPICH-Phoenix.

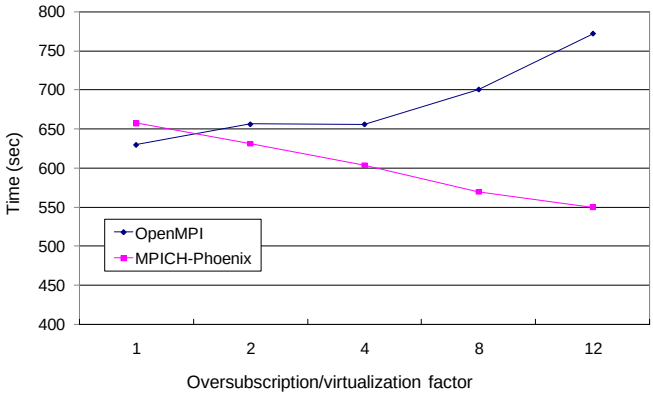

Figure 4. MILC runtime curves for OpenMPI and MPICH-Phoenix.

Fig. 4 shows the runtime for MPICH-Phoenix and OpenMPI running the MILC QCD code across eight nodes of Abe with increasing virtualization. Increasing the virtualization factor results in improvement of performance for MILC over MPICH-Phoenix while performance degrades with OpenMPI. As discussed in section III, virtualization endows certain benefits, such as computation and communication overlap and cache blocking effects. OpenMPI is unable to exploit these benefits due to high context switch overhead and lack of a lightweight integrated scheduler required for efficient network progress.

Fig. 5 compares the MILC runtime on Abe for MPICH-Phoenix and AMPI. AMPI tracks the MPICH-Phoenix curve, reflecting improved performance with greater virtualization. Table I summarizes the total time spent in each of the major phases of MILC over both these MPIs as well as the total runtime (NERSC). The table shows that MPICH-Phoenix consistently outperforms AMPI for all phases. We believe this is due to the lack of an integrated approach for scheduling of compute and communication tasks within a node. Charm++ runtime instance determines the task schedule locally, i.e., per core rather than per node, leading to suboptimal performance. In contrast, the integrated scheduling of compute and communication threads using Phoenix across all cores results in improved performance.

B. Communication Scheduling and Overlap

Asynchronous communication progress is a requirement for effective overlap of communication and computation. The COMB benchmark [20] is used to quantify the capability of achieving true overlap by computing the availability of a processor to do useful work between dispatch and completion of communication. Parallel programming paradigms, such as MPI, provide explicit asynchronous communication calls (*MPI_Isend* and *MPI_Irecv*) to enable overlap. However, a large number of MPI implementations do not provide true asynchronous communication semantics, requiring an application to periodically poll the network to make progress.

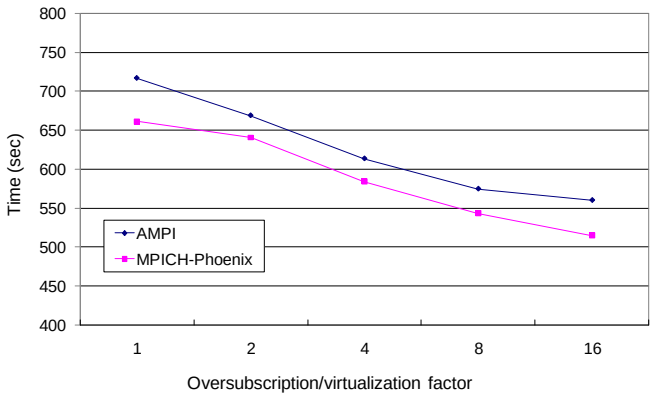

Figure 5. MILC runtime curves for AMPI and MPICH-Phoenix.

TABLE I. VARIOUS MILC PHASES AND TOTAL RUNTIME (NERSC) FOR AMPI AND MPICH-PHOENIX (ALL TIMES IN SECONDS)

Virtualization Factor	**1**		**2**		**4**		**8**		**16**	
MPI Flavor	***AMPI***	***MPICH-Phoenix***	***AMPI***	***MPICH-Phoenix***	***AMPI***	***MPICH-Phoenix***	***AMPI***	***MPICH-Phoenix***	***AMPI***	***MPICH-Phoenix***
LLTIME (Fat)	165.524	154.043	155.716	149.816	141.872	136.584	129.347	126.611	124.01	111.602
LLTIME (long)	10.2836	9.18277	9.3191	8.75081	8.3146	8.75402	7.15287	7.2549	6.0309	5.89462
GFTIME	173.333	164.734	156.596	154.263	136.917	136.503	121.656	123.854	115.177	106.593
FFTIME	251.915	232.175	233.742	232.545	216.281	206.245	203.412	184.334	194.72	178.628
CONGRAD5	101.565	86.6906	101.632	83.7191	99.2227	84.5697	103.156	91.0873	110.032	102.132
NERSC	717.204	661.471	669.125	640.982	613.656	583.891	574.73	543.05	560.115	514.507

Phoenix provides a rich infrastructure for experimenting with various communication scheduling strategies. A broad spectrum of communication scheduling strategies, from dedicating one or more cores for network communication to utilizing all available cores for compute and communication tasks, can be deployed. Fig. 6 depicts the availability of a processor performing asynchronous sends for various MPI implementations. Large message transfers typically employ a handshake protocol between sender and receiver. The Phoenix runtime schedules a network thread for execution on receipt of the handshake message, leading to high availability. Conversely, MVAPICH requires an application to poll the network to make progress, leading to minimal overlap.

Fig. 7 depicts the effects of various communication scheduling strategies for MILC execution on the Abe and QP clusters. The MILC NSF medium benchmark with 768 MPI tasks spanning eight nodes was used. Abe with eight processors per node had a virtualization factor of 12, while QP with four processors per node had a virtualization factor of 24. The *WQ* strategy utilizes all available cores for scheduling of computation and communication threads. The *Net Dedicated* strategy dedicates an entire core for communication handling. Finally, the *Net+WQ* strategy is a combination of both. In our current implementation of the WQ strategy, a communication thread is scheduled only in the absence of any compute threads being runnable. For each strategy, the total time a processor is idle (due to lack of any runnable tasks) as well as computing is captured. Additionally, we present the context switch overhead incurred due to virtualization of MPI tasks.

We observe that Phoenix can support fine-grained parallelism efficiently as the context switch overhead is negligible (< 0.1% of runtime) across all strategies. Both Net Dedicated and Net+WQ strategies result in reduced idle time compared to WQ. The WQ strategy is reactive, scheduling a communication thread only when a processor transitions to idle. We are evaluating the use of communication profile heuristics to implement anticipatory scheduling strategies.

The optimal strategy for execution differs between Abe and QP. On Abe the Net Dedicated strategy results in the fastest execution time while the WQ strategy offers superior performance on QP. Conventional MPI implementations are unable to support efficient execution in Net Dedicated mode without OS support, highlighting the benefits of our approach. Execution on QP is compute-bound and dedicating an entire core for communication dilates the compute time by 25% with a negligible reduction in idle time. On Abe, we can deduce that execution is network-bound. Abe nodes

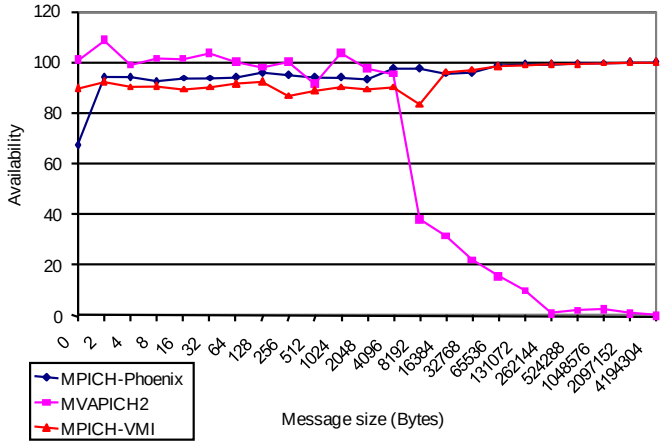

Figure 6. Availability metric for various MPIs using COMB benchmark.

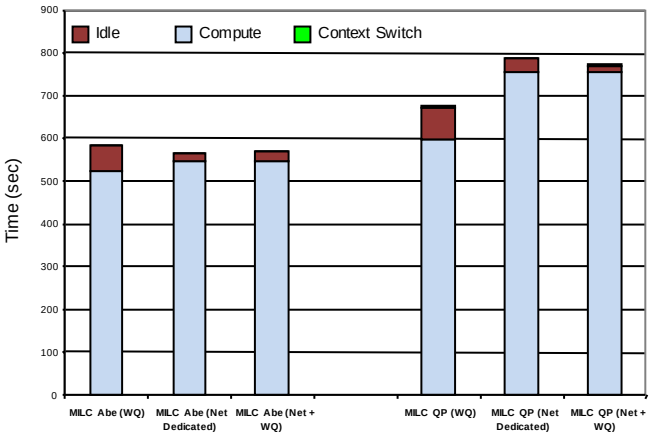

Figure 7. Integrated compute and communication scheduling for MILC.

consisting of eight processing cores are connected via a single data rate Infiniband network. With a dedicated core for communication, we observe a significant reduction in idle time due to network stalls, with a negligible increase in compute time as the virtualization factor increases from 12 to 14 per core.

C. *Effects of Cache Blocking*

The effect of cache blocking on MILC due to virtualization is considered. MILC is a popular open source quantum chromodynamics (QCD) application. The MILC NSF medium benchmark executing on 64 processors was used. Fig. 8 details MILC runtime in various routines. The *eo_fermion_force_3f* routine constitutes a significant portion of the total runtime and is the most compute-intensive memory bandwidth-bound MILC routine. With increasing virtualization factors, the MILC lattice can fit in the cache, reducing cache misses and improving time-to-solution. Fig. 9 shows the reduction in total runtime with increasing virtualization with a corresponding reduction in cache misses

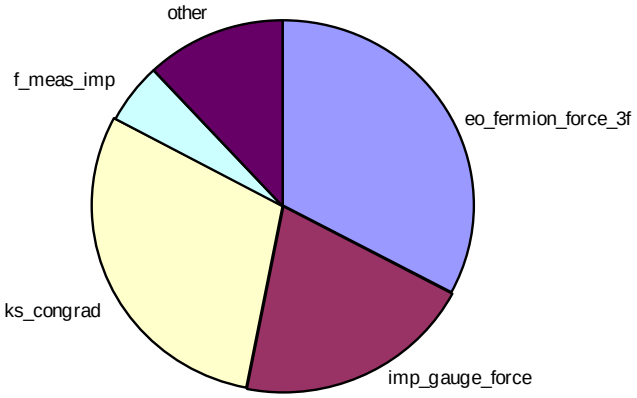

Figure 8. Distribution of MILC runtime while executing with 1024 tasks on 64 processors, 8 processors/node.

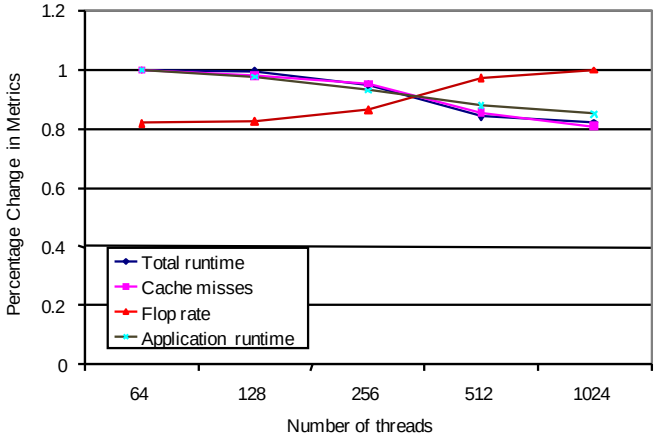

Figure 9. Various performance metrics for MILC fermion force calculation routine while executing on 64 processors, 8 processors per node.

and improved flop rate for *eo_fermion_force_3f*. We see close to 20% improvement in runtime of this routine and 15% improvement in runtime of the application.

V. Related Work

Our work on Phoenix leverages several efforts in the fields of operating systems, runtime libraries and high-performance computing in general. The Phoenix threading and synchronization subsystems are heavily influenced by user-level thread packages, such as pthreads, NPTL [21] and Marcel threads [22]. The Phoenix scheduler is similar in design to Intel McRT [10] and BubbleSched [23] but is significantly more configurable. McRT is designed to enable layering of parallel programming models for efficient execution on CMP. However, the McRT system does not itself consider the impact of incorporating communication scheduling semantics within the runtime. As we have shown, the selection of an appropriate network scheduling strategy is required for efficient execution of parallel codes on CMP architectures.

Various MPI implementations [4,5,9] exist that implement MPI tasks as lightweight threads. TMPI [5] is optimized for execution on shared memory machines. MPI tasks implemented as lightweight threads execute within a common address space, allowing for efficient communication. TMPI does not implement a virtualized runtime since the number of MPI threads is limited to the number of processors on the system. Thus virtualization benefits such as communication overlap and cache blocking cannot be exploited.

AMPI [9] and MPI_Lite [4] provide a virtualized MPI implementation. AMPI is implemented using the parallel Charm++ [8] library. Charm++ runtime, which provides the virtualization capabilities for AMPI, treats CMP architectures as a collection of SMP cores and there is no integrated scheduling capability across all cores of a CMP node, as we mentioned in the results section. Similarly, MPI-Lite requires the operating system to schedule the virtualized tasks. As we have shown, this results in increased overhead compared to user-level thread scheduling as well as limited flexibility in implementing application-specific scheduling strategies.

VI. Conclusions

Upcoming high-performance computing hardware will consist of nodes featuring chip multiprocessor cores. Current runtime systems view multicore hardware architecture as an array of SMPs. This view is fundamentally flawed. Crucial differences between CMP and SMP hardware, such as cache size and memory bandwidth, must be addressed in designing runtime systems for CMP-based hardware. We have proposed Phoenix, a runtime system designed specifically for CMP-based HPC nodes. We have shown that this system brings various performance and programmability benefits through fine-grained parallelism, virtualization, integrated scheduling, high-performance intranode communication and abstractions for layering parallel programming paradigms. Performance gains using micro-benchmarks and a

production application highlighted the benefits of our MPI implementation.

REFERENCES

[1] S. Gottlieb, W. Liu, R. Renken, R. Sugar, and D. Toussaint, "QCD thermodynamics with eight time slices," Phys. Rev. D, vol. 41-2, Jan. 1990, pp. 622-625.

[2] MIMD Lattice Computation Collaboration (MILC), http://www.physics.indiana.edu/~sg/milc.html

[3] Elsa: An Elkhound based C++ parser, http://www.cs.berkeley.edu/~smcpeak/elkhound/

[4] MPI-Lite, Parallel Computing Lab, University of California. http://may.cs.ucla.edu/projects/sesame/mpi lite/mpi_lite.html.

[5] H. Tang, K. Shen, and T. Yang, "Program Transformation and Runtime Support for Threaded MPI Execution on Shared Memory Machines," ACM Transactions on Programming Languages and Systems, vol. 22, Nov 2000, pp. 673-700.

[6] E. Lusk and W. Gropp, "The Second-Generation ADI for the MPICH Implementation of MPI", MCS Division, Argonne National Laboratory, 1996.

[7] E. Garbriel, et al., "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," Proc. 11th European PVM/MPI Users' Group Meeting, 2004.

[8] L. Kale and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," Proc. Conference on Object Oriented Programming Systems, Languages and Applications, Sept-Oct 1993

[9] C. Huang, O. Lawlor, L. Kale, "Adaptive MPI," Proc. 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), pp. 306-322, LNCS 2958.

[10] B. Saha, A. Adl-Tabatabai, R. Hudson, V. Menon, T. Shpeisman, M. Rajagopalan, A. Ghuloum, E. Sprangle, A. Rohillah, and D. Carmean, "Runtime Environment for Tera-scale Platforms," Intel Technology Journal, vol. 11, no. 3, 2007, pp. 207-251.

[11] TACC, Ranger supercomputer, http://www.rangersupercomputer.com/

[12] NCSA, Abe supercomputer, http://www.ncsa.uiuc.edu/

[13] Teraflops Research Chip, http://techresearch.intel.com/articles/Tera-Scale/1449.htm

[14] "MPI: A Message Passing Interface," Proc. Supercomputing '93, Nov. 1993, pp. 878-883.

[15] R. W. Numrich and J, "Reid: Co-Array Fortran for Parallel Programming," ACM SIGPLAN FORTRAN Forum, vol. 17, 1998, pp. 1-31.

[16] 7 Computational Dwarfs, http://view.eecs.berkeley.edu/wiki/Dwarfs

[17] G. Antoniu, L. Bouge, R. Namyst, "An efficient and transparent thread migration scheme in the PM2 runtime system," Lecture Notes in Computer Science, Parallel and Distributed Processing, Springer, Berlin/Heidelberg, vol. 1586/1999.

[18] UPC Language Specification, http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf.

[19] iWARP Consortium, http://www.iol.unh.edu/services/testing/iwarp.

[20] W. Lawry, C. Wilson, A. Maccabe, R. Brightwell, "COMB: A Portable Benchmark Suite for Assessing MPI Overlap," Proc. IEEE International Conference on Cluster Computing (CLUSTER'02), 2002, p. 472.

[21] U.Drepper, I. Molnar, "The native POSIX thread library for Linux," http://people.redhat.com/drepper/nptl-design.pdf.

[22] Marcel Thread Package, http://runtime.futurs.inria.fr/marcel/index.php

[23] S. Thibault, R. Namyst, and P. Wacrenier, "Building Portable Thread Schedulers for Hierarchical Multiprocessors: the BubbleSched Framework," Proc. EuroPar, Rennes,France, 2007.

[24] M. Showerman, J. Enos, A. Pant, V. Kindratenko, C. Steffen, R. Pennington, W. Hwu, "QP: A Heterogeneous Multi-Accelerator Cluster", unpublished.