Support for Adaptivity in ARMCI Using Migratable Objects

Chao Huang, Chee Wai Lee, Laxmikant V. Kalé

University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{chuang10, cheelee, kale}@cs.uiuc.edu

Abstract

Many new paradigms of parallel programming have emerged that compete with and complement the standard and well-established MPI model. Most notable, and successful, among these are models that support some form of global address space. At the same time, approaches based on migratable objects (also called virtualized processes) have shown that resource management concerns can be separated effectively from the overall parallel programming effort. For example, Charm++ supports dynamic load balancing via an intelligent adaptive run-time system. It is also becoming clear that a multi-paradigm approach that allows modules written in one or more paradigms to coexist and co-operate will be necessary to tame the parallel programming challenge.

ARMCI is a remote memory copy library that serves as a foundation of many global address space languages and libraries. This paper presents our preliminary work on integrating and supporting ARMCI with the adaptive run-time system of Charm++ as a part of our overall effort in the multi-paradigm approach.

1 Introduction

Parallel programming is certainly more difficult than sequential programming. The message passing paradigm, and MPI, a standard interface for it, is widely used. On certain shared memory machines, OpenMP is also used. However, it is believed that one needs to raise the level of abstraction in parallel programming to conquer its complexity, and to make it accessible for the large software community, whether in science and engineering applications or beyond.

"Raising the level of abstraction" connotes automating more of the functions a parallel programmer has to carry out. One direction that we are pursuing to this end is that of adaptive run-time systems empowered by a "migratable objects" (also called process virtualization) programming paradigm. The basic idea (as explained in section 2 below) is simple: require the programmer to express the decomposition (of data and work), and automate the mapping of data and work to processors and adaptive resouce management. In this approach, the programmer decomposes the computation into a large number of "objects" according to their logical meanings, regardless of the number of processors. The objects may be C++ objects, as in Charm++, or migratable user-level threads, as in Adaptive MPI. The number of objects is typically much larger than the number of processors; this creates a degree of freedom for the run-time system to assign the objects to processors, and to change this assignment as needed. Because the run-time system schedules the objects and mediates communication between them, it can automatically instrument their execution, and measure the computation and communication patterns. Since these patterns tend to persist in science and engineering applications, accurate measurement-based load balancing becomes possible. Thus, this approach raises abstraction level by automatic run-time optimizations that the programmer would otherwise have carried out explicitly.

Other ways of raising the abstraction level involve co-ordination and information-sharing mechanisms among processes. MPI-style message passing is a primitive method of coordination. Charm++ provides asynchronous method invocation as another, also primitive but distinct, coordination mechanism. In addition, paradigms based on global address space such as Global Arrays[19], UPC[5], Co-array-Fortran[20] provide a data-sharing mechanism that can simplify programming for at least some classes of applications and algorithms. Multi-phase shared arrays[3] is a model we developed that attempts to derive the expressiveness benefits of shared address space, while limits the kind of sharing that can be done so that it can be efficiently supported without race conditions. Other languages, libraries and programming paradigms such as HPF (High Performance

This work was supported in part by DOE Grant B341494 and B505214, and by the National Science Foundation through TeraGrid resources at NCSA.

Fortran)[15], ZPL[17], BSP[25] and even Linda[7], provide a wide variety of programming abstractions.

Each of these models are useful and expressive in specific contexts. Often one can find example applications or application kernels where one of the paradigms is a perfect fit. Ideally, we would like to support multiple parallel programming paradigms in a single application. For example, each module may be programmed in a paradigm most suited to it, or one that its programmer is most proficient in. Alternatively, some modules may be programmed using multiple programming paradigms.

Such multiparadigm programming must be supported by *concurrent composibility*: the ability to automatically interleave the execution of modules such that idle time in one can be overlapped by useful computation in another. Without this, one either loses performance or (more likely) encourages programmers to break abstraction boundaries for the sake of performance. For this reason, as well for the mere ability to allow co-existence of multiple paradigms in a single application, the paradigms (languages and libraries) must share a common run-time. A common run-time can also provide common functions that are needed across multiple paradigms, such as checkpointing support.

As a result, we are pursuing an approach where a common run-time based on migratable objects and user-level threads is used to support multiple parallel programming paradigms simultaneously. This combines the benefits of our adaptive run-time system, the concurrent composibility induced by message-driven execution in the run-time system, and benefits of multi-paradigm programming (i.e. the ability to choose the best paradigm for each module separately, for example). We have already implemented Charm++ and Adaptive MPI in this framework, along with some other mini-languages such as MSA[3] and structured-dagger.

This paper outlines our preliminary work on integrating global-address-space programming models within this framework. We select ARMCI, which is a foundation of some other global-address-space approaches such as GA[19], Co-Array Fortran Compiler[4] and Adlib[2], for our first implementation. We will show how we "virtualized" (i.e. supported via our virtual-process approach) ARMCI, and what benefits derive from such an implementation. We begin with a description of benefits of virtualization and a review of ARMCI and the programming model it engenders.

2 Adaptive Run-Time System

In the programming paradigm supported by the Adaptive Run-Time System (ARTS)[11], the programmer divides the parallel program into a large number of partitions, independent of the number of physical processors. The partitions of work, which can be viewed as virtual processes (VPs), are then implemented using migratable objects and mapped onto the processors by the system. The programmer, not being constrained by physical processors, is able to focus on the interaction between the work partitions and better expression of the parallel algorithm. On the other hand, the ARTS can perform efficient resource management with the large number of migratable objects (VPs). In this section we highlight several of the many benefits that accrue from this programming paradigm with ARTS (See Figure 1).

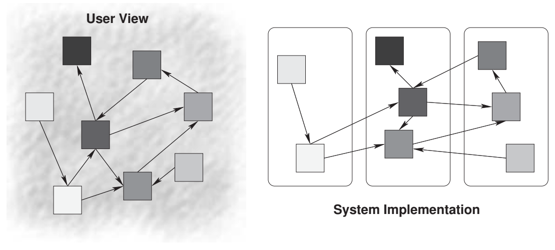


Figure 1. Adaptive Run Time System with Migratable Objects

2.1 Adaptive Overlap

Consider the scenario illustrated in Figure 2 (This example is taken from [11]). There are three parallel modules A, B and C spread across all processors. A must call B and C, but there is no dependence between B and C. In traditional MPI style programming model, the programmer has to choose one module between B and C to call from A first on all processors. Only when the first chosen module returns can A call the remaining one on all processors. This model can be inefficient because when one module idles the CPU, for example, when waiting for communication to complete, other modules are not allowed to take over and do useful computations, even though there is absolutely no dependence between the modules.

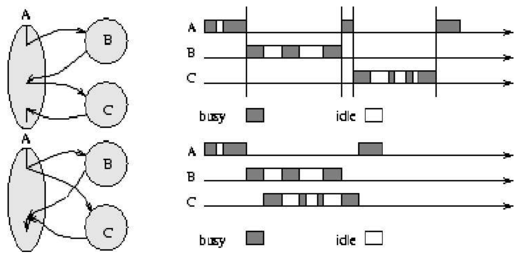


Figure 2. Adaptive Overlapping

With ARTS support, A can invoke B on all the VPs, initiating computation and sending out messages, and since there is no dependence between B and C, A can also start off C in a similar fashion, and thereby modules B and C can interleave their execution. When one module blocks due to communication or load imbalance, the other module can automatically overlap the idle time with computation, based on the availability of data, as illustrated in Figure 2. With non-blocking calls and careful programming, the programmer could achieve the same effects with MPI, but the price is additional programming complexity and a breach of modularity.

2.2 Automatic Load Balancing

One of the most prominent benefits of ARTS with migratable objects is the capability of dynamic load balancing by migrating objects across processors. The challenge is for the ARTS to intelligently determine an effective remapping of objects for the future. Before describing our load balancing scheme, we introduce an important observation called the *principle of persistence*. Like the principle of locality, the principle of persistence is an empirical heuristic about parallel program behaviors. We observe that for most parallel programs expressed in terms of VPs, the computation loads and communication patterns tend to persist over time. This heuristic applies to many programs with dynamic behavior, including those using adaptive mesh refinement with abrupt but infrequent changes, and those simulating molecular dynamics with slow and gradual changes over time.

Based on the principle of persistence, our ARTS uses a measurement based load balancing scheme. The load balancer automatically collects statistics on each object's computation loads and communication patterns, and using the collected load database, the ARTS decides on when and where to migrate the objects following a load balancing strategy. A variety of such strategies have been developed for applications with different dynamic behaviors. Some strategies are centralized, others fully distributed. Some use only computation load when making a decision, and others take into account communication patterns and even topology of the platform. Our previous work on NAMD[14] demonstrates the significant benefits of automatic load balancing in real-life applications.

2.3 Automatic Checkpointing

Checkpointing for an application run on an ARTS is as simple as migrating the objects (VPs) onto the target media: either hard disk drive[9] or memory on peer nodes[27]. It is important to note that the checkpoint/restart mechanism in the ARTS has benefits beyond fault tolerance. Imagine if we lose 1 node out of a 1024-node partition in the middle of a long execution. We can immediately work around this failure and restart the checkpointed program, with the same number of VPs, but on 1023 physical processors. This concept can be extended to a shrink/expand feature, which allows an adaptive application to shrink or expand the set of physical nodes on which it runs at run time.

2.4 Communication Optimizations

The Charm++ ARTS is capable of observing the communication patterns, and consequently able to optimize communication performance by switching different communication algorithms automatically. For instance, if we keep track of information like the number of physical processors and VPs involved in collective communications and the amount of data transferred through the links, we are able to choose the best suited collective communication strategy. Research work is being done to make the ARTS "smarter", able to dynamically learn and shift to the most suitable strategy.

2.5 Software Engineering Benefits

An ARTS with migratable objects helps programmers to practice good software engineering disciplines, such as high cohesion and low coupling. High cohesion means any module in a program should be understandable as a meaningful unit and components of a module should be closely related to one another. Low coupling requires that different modules be understandable separately and have low interaction with one another. With a traditional processor-centric programming model like MPI, it is often almost inevitable that programmers will violate these principles; whereas with virtualized processes, programmers are given the freedom to partition and structure the parallel application in accordance with good software engineering principles. For example, a physical simulation is performed on a structured cube-shaped grid. Programming traditional MPI typically involves multi-blocking part of the grid onto one processor, sacrificing the cohesion of the program. In the ARTS, programmers can partition the work on a cubed number of virtual processes that can naturally express the algorithm, and lets the ARTS efficiently manage resources such as physical processors and interconnect.

3 Virtualized Multi-Paradigm Parallel Programming

As suggested in the introduction, message-driven execution creates an opportunity for effective multi-paradigm programming. The common scheduler on each processor, which is needed anyway to handle multiple work units (objects in Charm++) assigned to the processor, allows it to to automatically interleave execution of modules written in different paradigms. Of course, this requires that each paradigm be implemented the common run-time system. Interoperability of multiple paradigms itself compels such a common run-time system.

To this end, we have broadened the run-time system in Charm++. In fact, the common components needed in the run-time system were separated early on into a layer called Converse[12]. Converse provides a machine independent interface to the capabilities offered by a parallel machine and the local operating system. In addition to communication, this includes an implementation of non-preemptive user-level threads.

Typical thread packages combine functionality of creation of threads, scheduling of threads, and synchronization between them. Converse's threads factor out this functionality into separate components, to allow the run-time system direct control over scheduling. The core thread layer only provides the capability to encapsulate a stack and current set of registers, including the program counter, and the ability to switch contexts from one thread to the other. The Converse (and therefore Charm++) scheduler manages threads. In other words, the scheduler's queue includes a general form of "messages" each with its own handler, and the ready threads simply are a special case of such messages. This design has proved to be extremely flexible to support many different paradigms and libraries.

Charm++'s abstraction of "object-arrays", which is a collection of objects indexed by any general index structure (including strings, bit-vectors and multi-dimensional sparse index sets), turn out to provide a basic functionality needed by most programming paradigms. For example, MPI processes are indexed by their rank; so implementing them using chare-arrays allows us to reuse the code for locating objects, redirecting messages to them after migration (if necessary) and maintaining tables of known-locations. As a result, Charm++, rather than Converse, is used as a common layer on which to implement other languages and libraries. This also simplifies implementation of load balancing and other run-time functionalities.

The resultant architecture we use (and plan to use further) for multi-paradigm programming is shown in Figure 3. The Charm++ run-time system is directly used to implement language and library run-times. For example, in our MPI implementation, each MPI "process" is implemented as a user-level thread embedded inside a Charm++ object. When it sends a message to "process x", the call goes to Charm++ run-time, which sends the message to Chare x of the appropriate object array, where it is delivered to the MPI run-time.

In this architecture, common services are implemented via callbacks to the RTS's of specific paradigms (and to the application). For example, when the common adaptive RTS decides to migrate an entity, it uses a callback into the paradigm's RTS to ask it to pack-up the entity and then installs it on the remote processor chosen by the common RTS. Check-point and restart are also implemented by similar callbacks. As a result, with a small effort while porting the RTS of a paradigm to the framework, one can easily provide such adaptive functionalities to the new paradigm. Thus, for example, extending an MPI implementation (in isolation) to have the ability to shrink and expand the sets of processors used by an MPI application may require much effort, but this becomes quite easy if one leverages the capabilities of the common Adaptive RTS.

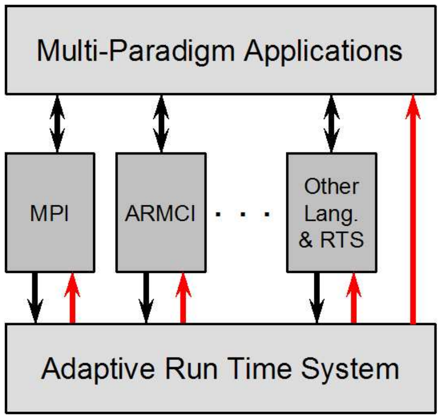


Figure 3. Architecture of Adaptive Run-Time System Supporting Multi-Paradigm Programming

3.1 Charm++ and TCharm

Charm++[13] is our implementation of an ARTS. It is a portable C++-based programming language. In Charm++, migratable objects are known as *chares* and are implemented as C++ objects with special *entry methods* that are invoked asynchronously from other chares. Charm++ makes use of run-time schedulers to determine which chare gains control on a processor via *message-driven execution*. In employing the technique of message-driven execution, the key feature of adaptive overlap in ARTS is achieved as no chare can hold a processor idle while it is waiting for a message. Charm++ has also been developed with many features of a full-fledged ARTS in mind, in particular a powerful automatic load balancing framework [26], adaptive communication optimization framework [16], and support for multiple fault-tolerance and recovery schemes[9, 27, 23].

A number of major real-world scientific applications are implemented using Charm++. NAMD[21], a parallel classical molecular dynamics application was awarded the Gorden Bell Award, having achieved 1.08 TFlops scaling to 3000 processors[22] on Pittsburgh Supercomputing Center's Lemieux machine. LeanCP[24] is an application that employs the Car-Parrinello ab initio molecular dynamics method and is the result of a close collaborative effort. ParallelGravity[8] is a scalable cosmological simulation used for the study of formation of galaxies and planetary systems.

Threaded Charm++ or TCharm is a framework built on top of Charm++ that provides common run-time support for migratable and light-weight threads. When another framework needs thread support, the programmer simply "binds" the VPs from that framework onto a set of TCharm threads. The virtual processes then use the bound thread as needed, and the ARTS always migrates the bound VP and thread together. TCharm also provides language-neutral facilities for combining multiple application frameworks within a single program. This is achieved by allowing VPs from multiple frameworks to be attached onto the same set of TCharm threads. For example, one can use TCharm to create a Finite Element Framework application that also uses Adaptive MPI to communicate between Finite Element chunks.

3.2 Example: Adaptive MPI

Adaptive MPI (AMPI) [10] is our effort to integrate and support MPI with the ARTS. AMPI implements migratable virtual processes, several of which can be assigned to one physical processor. This efficient virtualization provides a number of benefits described in Section 2, and is widely portable. AMPI started as a proof-of-principle project to demonstrate that message passing models can be effectively supported in an ARTS such as Charm++; by now AMPI is already a mature system that can be and has been used in applications, especially those with dynamic nature, and thus can benefit from its adaptive features. Our goal is to achieve the same benefits for the adaptive implementation of ARMCI, and thereupon support some of the important global address space languages and libraries with Charm++ ARTS.

4 Implementation of ARMCI

ARMCI[18] is a multi-platform library for high-performance remote memory copy. It provides an interface for operations in the following three categories:

- data transfer operations including put, get and accumulate, in both blocking and nonblocking modes
- synchronization operations - local and global fence and atomic read-modify-write, mutex operations
- utility operations for allocation and deallocation of memory and error handing

ARMCI has been used in several global address space languages and parallel distributed-array libraries and compiler run-time systems including Global Array[19] and Co-Array Fortran Compiler[4]. Compared with traditional message passing paradigm, ARMCI has a few advantages in programming models that deal with distributed arrays.

Firstly, ARMCI's one-sided communication paradigm separates data transfer from synchronization between the process that needs the data and the process that owns the data. MPI-2[6] does provide one-sided communication interface, but its active target synchronization "requires the user to provide complete information on the communication pattern, at each end of a communication link." This means the users are expected to provide more information than needed by the underlying RDMA calls. One-sided operations in ARMCI, however, do not require explicit cooperation on the remote process, and hence significantly simplify implementation of parallel algorithms accessing distributed and irregular data structures remotely.

Secondly, ARMCI provides a well designed interface for non-contiguous data accessing that generalizes data transfer patterns of typical distributed array operations found in scientific applications. The vector interface expresses gather and scatter patterns well, and the strided interface naturally specifies copying a section of multi-dimensional arrays.

The following subsections describe our implementation of ARMCI.

4.1 Virtualizing ARMCI Processes

Virtual ARMCI processes are implemented with user-level TCharm threads embedded in migratable objects. Each VP encapsulates the state of an ARMCI process required for operations, such as the memory pointers maintained remote copy. As described in Section 3.1, each VP is bound to a user-level thread so that they always migrate together during load balancing.

4.2 Memory Allocation via Isomalloc

ARMCI provides a collective memory allocation scheme for use with copy operations. The collective call returns to every ARMCI VP an array of pointers to the newly allocated memory on each ARMCI VP. The user then uses these pointers to determine the memory locations for copy operations. In general, to support adaptivity and migration under this scheme for memory allocation, the system would have to broadcast the new pointer locations for each allocated memory block of each migrated ARMCI VP to every other ARMCI VP.

We have chosen to implement this memory allocation scheme using isomalloc [1]. Isomalloc is a technique for allocating heap memory that resides in the same virtual address location on each processor. Each VP is effectively given a band of the virtual address space from which heap memory can be allocated. In such a scenario, when a VP is migrated to a new processor, the collectively allocated memory is copied into the same virtual memory address space and their respective pointers need not change.

Isomalloc makes this collective memory allocation easy to maintain even in the presence of migration; there is no need to update the pointer values stored on all other ARMCI VPs when one VP migrates to a new processor. Of course, a major limitation of this technique is the amount of virtual memory space available to the heap of each VP. We expect, however, that the general move to 64-bit addressing in most modern machines would allow this technique to work effectively, even for supporting a very large number of VPs.

4.3 Remote Memory Copy through Threaded Message-Driven Execution

Remote memory copy is implemented as Charm++ messages to other VPs. The message-driven execution model allows for a natural implementation of non-blocking calls. Blocking calls are implemented by suspending the calling TCharm thread until a response is received from the target thread.

The underlying Charm++ run-time is also now in a position to intelligently manage messages to optimize the resulting communication between VPs. For example, if a piece of data is intended for another VP that also resides on the same processor, that data can merely be copied rather than pushed through the Charm++ message scheduler.

5 Performance Evaluation

In this section we show some preliminary results from our benchmarks. Our main benchmarks are the ones included in the ARMCI distribution: `perf.c` for contiguous and strided copy performance, and `lu.c` that implements LU algorithm and `lu-block.c` for LU block distribution. All of our experiments are performed on NCSA's IA-64 TeraGrid Cluster with 888 dual Intel Itanium 2 nodes and Myrinet network.

5.1 Virtualization Overhead

In this section we show the overhead incurred by virtualization. Because our adaptive implementation of ARMCI uses messages to implement memory copy, and our current implementation does not make full use of the native RDMA mechanism available, we do not expect to have better performance than the native implementation now. The short message latency of the adaptive implementation is 28μs for get and 27μs for put, about 12μs slower than the native implementation. This is due to an extra 70 byte ARMCI message header and a 2-4 microsecond increase in thread context switch overhead as well as scheduling overhead. For long messages, data in Figures 4 and 5 show that we pay the overhead of extra message copying in order to support migratable objects. Ongoing work is aimed at reducing the overhead for both scenarios, and it is more important to note that adaptive implementation is expected to outperform native implementation when its features such as automatic adaptive overlap and dynamic load balancing are utilized in real applications.

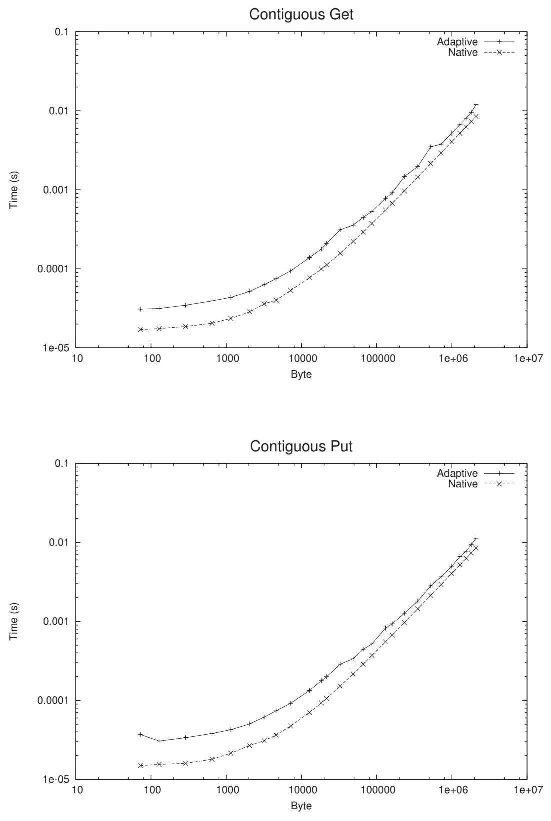
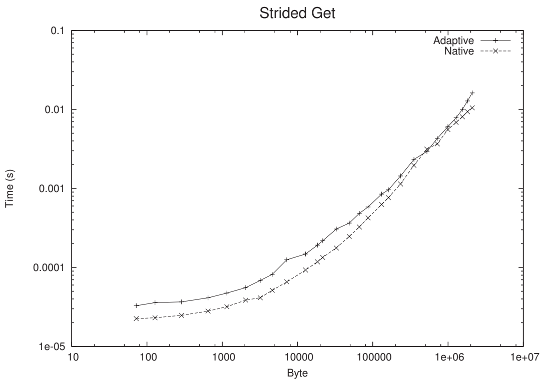


Figure 4. Contiguous Copy Performance of Adaptive and Native Implementations

5.2 On-Disk Checkpoint/Restart

Charm++ run-time system includes a checkpoint/restart mechanism[9] as an effort toward fault tolerance. It provides the user with the capability to take snapshots of an ARMCI program, either periodically or on command. The



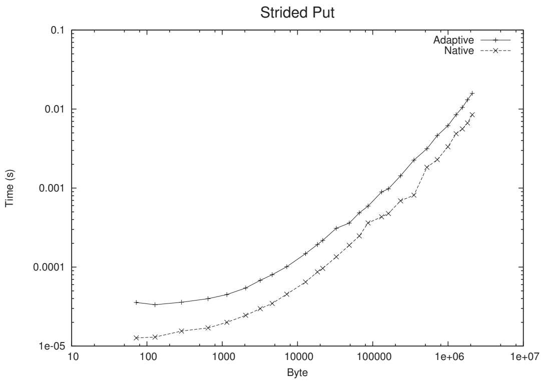


Figure 5. Strided Copy Performance of Adaptive and Native Implementations

checkpoint mechanism is able to intelligently, sometimes with guidance from the user, save only the data essential to resurrection of the program on occurrence of system failure.

P	Total Data (MB)	Time (ms)	Bandwidth (MB/s)
2	20.05	221	90.8
4	22.29	249	89.7
8	26.50	303	87.6
16	35.43	366	96.9
32	53.27	533	100.0

Table 1. Checkpoint Overhead

Data in Table 1 shows the on-disk checkpoint overhead of the LU application written in ARMCI, on 2 to 32 physical processors. The total amount of data varies, but we observe that the total bandwidth for disk I/O does not scale. This implies that in this experiment setting, the bottleneck is in accessing the network file system. If the machine allows the program to checkpoint to local disk, the checkpoint overhead should decrease almost linearly with the number of processors increasing, as shown in previous work[9].

To avoid disk I/O bottleneck, the programmer can choose the in-memory checkpoint mechanism[27], which logs the checkpointed data in memory on peer processors. This alternative is advantageous on platforms where NFS disk I/O is limiting factor, especially for applications with a moderate memory footprint. Performance evaluation of the in-memory checkpoint approach for ARMCI applications will be carried out in the future.

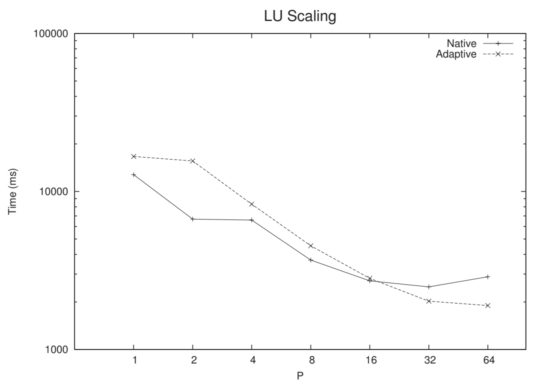
5.3 LU Performance

Figure 6 visualizes some preliminary results from performance scaling runs of the LU and LU-Block application included in the ARMCI distribution. The Adaptive runs are done with the number of VPs chosen to give the best performance (in this particular case, typically with 32 or 64 VPs). From the figure we observe that the adaptive implementation, although starting off slower than the native implementation, continues to scale well until the point where the native implementation cannot. While the exact reason for the performance of the native implementation to deteriorate exceptionally at larger numbers of processors is yet to be investigated, this benchmark shows that the adaptive implementation is able to perform comparably, if not better, without even taking advantage of other features like automatic load balancing and communication optimization.

6 Discussion and Conclusion

We have presented an adaptive implementation of ARMCI on top of Charm++. We implement virtual ARMCI processes on migratable objects bound with light-weight user-level threads. Several of these virtual processes can be mapped to one physical processor. This efficient virtualization provides a number of benefits, including the ability to automatically overlap computation and communication, automatically load balance arbitrary computations, emulate large machines on small ones, tolerate faults at the system level, and respond to a changing physical machine.

Adaptive implementation of ARMCI is an active research project. As planned future work, we will further optimize the point-to-point performance by shrinking message header and using RDMA operations where available. We will also help port global address space languages and libraries, such as Co-Array Fortran Compiler onto our implementation. We then plan to explore applications that use ARMCI that can benefit from automatic dynamic load balancing.



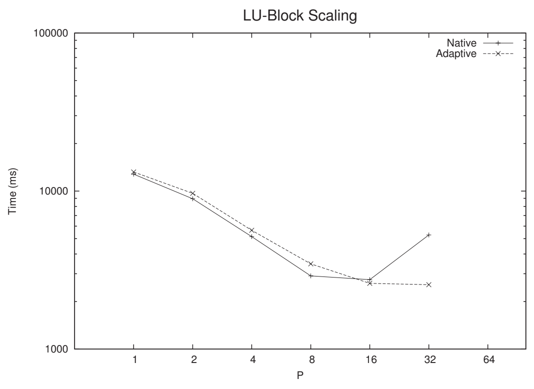


Figure 6. Performance Scaling of Adaptive and Native Implementations

References

[1] G. Antoniu, L. Bouge, and R. 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.

[2] B. Carpenter, G. Zhang, and Y. Wen. NPAC PCRC runtime kernel definition. Technical Report CRPC-TR97726, Center for Research on Parallel Computation, 1997.

[3] J. DeSouza and L. 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.

[4] Y. Dotsenko, C. Coarfa, and J. 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] T. El-Ghazawi and F. Cantonnet. Upc performance and potential: a npb experimental study. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–26, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[6] M. P. I. Forum. MPI-2: Extensions to the message-passing interface, 1997. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[7] D. Gelernter, N. Carriero, S. Chandran, and S. Chang. Parallel programming in Linda. In *International Conference on Parallel Processing*, pages 255–263, Aug 1985.

[8] F. Gioachin, A. Sharma, S. Chackravorty, C. Mendes, L. V. Kale, and T. R. Quinn. Scalable cosmology simulations on parallel machines. In *7th International Meeting on High Performance Computing for Computational Science*, July 2006.

[9] C. Huang. System support for checkpoint and restart of charm++ and ampi applications. Master's thesis, Dept. of Computer Science, University of Illinois, 2004.

[10] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé. Performance evaluation of adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

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

[12] L. V. Kale, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.

[13] L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[14] L. V. Kalé, S. Kumar, G. Zheng, and C. W. Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop, International Conference on Computational Science(ICCS)*, Melbourne, Australia, June 2003.

[15] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[16] S. Kumar. *Optimizing Communication for Massively Parallel Processing*. PhD thesis, University of Illinois at Urbana-Champaign, May 2005.

[17] C. Lin and L. Snyder. ZPL: An Array Sublanguage. In *Languages and Compilers for Parallel Computing (Proceedings of the Sixth International Workshop)*, pages 96–114. Springer-Verlag, 1994.

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

[19] 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.

[20] Numrich and Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, 1998.

[21] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[22] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.

[23] Sayantan Chakravorty, Celso Mendes and L. V. Kale. Proactive fault tolerance in large systems. In *HPCRI Workshop in conjunction with HPCA 2005*, 2005.

[24] R. V. Vadali, Y. Shi, S. Kumar, L. V. Kale, M. E. Tuckerman, and G. J. Martyna. Scalable fine-grained parallelization of plane-wave-based ab initio molecular dynamics for large supercomputers. *Journal of Comptational Chemistry*, 25(16):2006–2022, Oct. 2004.

[25] L. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8), August 1990.

[26] G. Zheng. *Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[27] G. Zheng, L. Shi, and L. V. Kalé. Ftc-charm++: An in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *2004 IEEE International Conference on Cluster Computing*, San Dieago, CA, September 2004.