Communication Support for Dynamic Load Balancing of Irregular Adaptive Applications*

Andriy Fedorov and Nikos Chrisochoides
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23185-8795
{fedorov,nikos}@cs.wm.edu

Abstract

In this paper we present a runtime system (Clam) that provides support for one-sided communication, remote service request, global address space in the context of mobile user-defined data objects and transparent routing of messages to those objects. We describe the functionality of Clam, the motivation and major design decisions behind its implementation. The functionality provided by Clam proved to be useful for dynamic load balancing support of adaptive asynchronous and irregular applications, such as mesh generation. The design of Clam is based on our experience with the previous implementation of the PREMA communication subsystem. We found it necessary to re-evaluate the design priorities set originally. In this paper we investigate these design decisions and evaluate their impacts on the runtime system.

1. Introduction

In this paper we present the design, implementation and evaluation of *Clam*, a light communication layer for asynchronous irregular and adaptive computations. *Clam* is a runtime system that addresses the computation and communication requirements of applications like parallel Adaptive Mesh Refinement (AMR). Our approach is based on a careful balancing of the following issues: correctness, performance, and ease-of-use. The preliminary results show that the implementation is portable, intuitive, and introduces low overheads over the underlying communication substrate.

The computations for adaptive applications like AMR can be tightly or partially (loosely) coupled, or they can be decoupled. Partially coupled applications can postpone processing of the incoming messages without delaying the computation. Tasks within tightly coupled computations communicate intensively and may have to wait for incoming communication, or suspend until the previously posted message is acknowledged by the partner task. Decoupled computations have no dependencies between the parallel tasks. The AMR applications also exhibit irregularity of the workload and require dynamic load balancing. From our experience, direct use of the existing low-level communication libraries is not desirable for such applications because of high development and maintenance costs.

The Portable Runtime Environment for Mobile Applications (PREMA) framework has been created to support development of AMR-like applications. Although the functionality provided by *Clam* can be used directly by the application, its main purpose is to serve the needs of the Implicit Load Balancing library (ILB) [7] within PREMA. As the communication component of the PREMA framework *Clam* is superior to the previously used implementations [8, 12], which is evident from our analysis and evaluation. We improved upon the previous system based on the lessons learned from supporting and using that system for development of mesh generation applications.

The main contributions of this paper are: (1) design of a robust runtime system, which is based on (2) analysis of the problems within the previous implementation of the PREMA communication layer and (3) re-balance of the correctness, performance and ease-of-use trade-offs. We also reconsider the correctness responsibilities of the application and the runtime system.

The paper is structured as follows. Section 2 surveys the previous work on communication support within PREMA and motivates the new implementation. In Section 3 we describe the functionality, design and some implementation details of the runtime system. Section 4 presents our experimental evaluation. We conclude with the discussion and directions for future work in Section 5.

*This work was partially supported by the National Science Foundation grants ITR-0312980, NGS-0203974, ACI-0085969, EIA-9972853 and CCR-0049068.

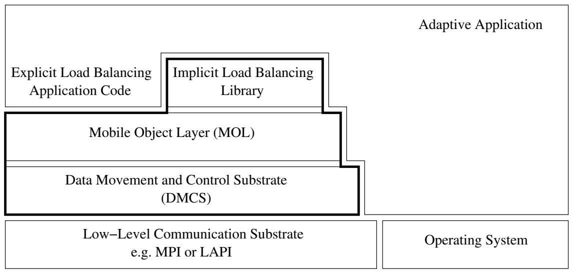
Adaptive Application
Explicit Load Balancing Application Code
Implicit Load Balancing Library
Mobile Object Layer (MOL)
Data Movement and Control Substrate (DMCS)
Low–Level Communication Substrate e.g. MPI or LAPI
Operating System

Figure 1. The initial architecture of PREMA.

2. Related Work and Motivation

The original implementation of the communication layer for PREMA consisted of the Data Movement and Control Substrate (DMCS) [8] and the Mobile Object Layer (MOL) [12]. DMCS provided one-sided communication and remote service request (RSR) functionality atop low-level communication primitives of MPI or LAPI. MOL implemented global address space for so called *mobile objects*. A mobile object in MOL is a user-defined data structure which can be migrated from one processor to another and referenced independently of its current location. Throughout the paper we refer to the initial implementation of the PREMA runtime layer as the DMCS/MOL implementation.

The mobile object functionality serves as the basis for the Implicit Load Balancing Library (ILB) [7], which uses mobile object abstraction to implement *schedulable objects* (SOs). A SO is the smallest unit of granularity managed by the ILB. Using RSRs, the ILB module collects information about the load distribution within the system. If the imbalance is detected, local SOs can be migrated. The initial structure of PREMA is shown in Figure 1.

It has been shown that the functionality provided by PREMA has a number of advantages over similar load balancing systems [6]. Based on the experience within our group, PREMA greatly simplifies the application design by separating low-level systems and load balancing issues from the developer [5, 7].

Clam has been designed to serve as the new communication layer for the PREMA framework. Below we discuss some of the problems we discovered within the DMCS/MOL implementation. For a comprehensive survey of related work in the context of other runtime systems the reader is referred to [5, 7].

Despite the initial intentions, the DMCS/MOL implementation failed to become independent of low-level communication substrate. The implementation we were using was built on top of MPI. This became problematic since one of our projects requires running PREMA on top of Globus [1]. The problems we encountered [3, 4] with the Globus-based implementation of MPI (MPICH-G2) turned out to be difficult to overcome and have not been resolved in the two years since our initial reports [2]. This was the primary motivation for the communication subsystem redesign: to break the dependency on MPI. However, in the process of the development of *Clam* we made a number of other design decisions in order to consider the following problems identified in the DMCS/MOL implementation:

1. lack of interoperability (MPI, C++, STL);
2. poor usability (duplication of functionality, overloaded API);
3. DMCS/MOL “separation of concerns”: performance issues;
4. complicated maintenance because of 1-3;
5. insufficient correctness guarantees (communication deadlock).

3. Design and implementation issues

3.1. Functionality

Computation and communication volumes and patterns for adaptive and irregular applications like AMR are variable and unpredictable. One-sided communication substantially simplifies code development and maintenance for such applications. The functionality of *Clam* can be grouped as follows:

- **remote memory operations**: *put* and *get*;
- **remote service request (RSR)**: invocation of an application-defined function on a remote processor;
- **mobile object functionality**: creation, migration, and communication with mobile objects.

The *Clam* runtime system supports SPMD model; each processor is assigned a unique identifier maintained by *Clam*. A user-defined set of functions, *handlers*, is registered with the runtime system. A handler can be invoked remotely. There are four types of handlers: (1) memory operation, (2) RSR (fixed number of arguments), (3) RSRN (RSR that takes a buffer as an argument) and (4) mobile object message handlers.

The communication operations provided by *Clam* are non-blocking and asynchronous. The RSR functionality is intended to support single-sided *processor-to-processor* communication. This is required for: (1) load imbalance information dissemination, (2) load redistribution, and (3) application communication needs.

An application can associate what we call a *mobile pointer* with any data object local to the processor's memory. We also call the data structure which has a mobile pointer associated with it a *mobile object* (we use these terms as they were introduced earlier in [12]). The concept of mobile pointers provides *processor-to-data* communication. The data structure associated with a mobile pointer can be migrated to any processor following predefined *Clam* procedures. While the RSR functionality allows an application to invoke a handler on the specified processor, mobile pointers allow the application to invoke a handler on a processor where the targeted mobile object is currently located. A request for such invocation is translated into a *message* to the mobile object. The runtime system provides location-independent routing of such messages to mobile objects.

The mobile object functionality provides support for application adaptivity. The workload local to a processor can be represented by the set of data objects in memory of that processor (this is a particularly useful abstraction for AMR applications). During load balancing, a local work-unit (object) can migrate to the address space of a remote processor. The computation is not decoupled in the general case: there may be dependencies between work-units located on different processors. Hence the requirement for the processor-to-data communication support.

Clam implements the single-threaded execution model: all handlers are executed during explicit polling, waiting on barrier, or waiting for global communication completion (quiescence detection [14]). Although the targeted applications are asynchronous, the synchronization and quiescence detection operations provided by *Clam* are critical for the initialization and termination detection procedures.

3.2. Design

In this section we identify and elaborate on a number of design issues which we consider the most important within the runtime system. As we show, there are multiple trade-offs which we have to take into account.

Correctness *Clam* implements one-sided communication model with explicit polling. One of the most desired correctness requirements is to provide deadlock-free communication within this model. In the send/receive communication model deadlock prevention is the responsibility of the application. In Active Messages (AM) [16, 20], deadlocks are avoided by restricting the application communication freedom. We find both send/receive and AM models too restrictive to implement dynamic load balancing and AMR applications. The only restriction which *Clam* puts on the application is that the poll operation may not be invoked within a handler. This flexibility leads to a possibility of memory exhaustion, thus a possibility of deadlock.

The deadlock prevention responsibilities are shared between the runtime system and the application code. It is absolutely necessary to have the correct deadlock-free implementation of the application. In our case, an application bears the responsibility of careful use of *Clam* communication functionality which will not lead to deadlock. While it is not feasible to completely eliminate the possibility of deadlock in *Clam*, we consider it important to avoid it as much as possible by using only the non-blocking low-level communication functions within the runtime system.

Clam guarantees error-free data transmission, FIFO per-processor ordering of incoming handlers execution and ordering of messages sent from a processor to the object.

Performance The runtime system overhead should be low compared to the actual communication time. Also, with changes in the configuration size, the ratio of runtime system overhead to the total communication overhead on a particular processor should not change significantly, i.e., the system should be scalable.

The runtime system performance is dependent on the decisions made relative to the correctness and safety of the implementation. As we mentioned above, it is important to use non-blocking communication within the runtime system. Because of this, additional data structures are required to keep track of pending communication operations. Also, blocking communication operations are known to be the most efficient. The non-optimal communication mode is the necessary price to pay for the desired level of correctness.

Another issue which affects performance is the number of intermediate layers within the implementation. From one hand, the "separation of concerns" principle simplifies the logical view of the implementation and its maintainability. However, it is possible that the overheads introduced by the separation can be too high to justify the benefits. In *Clam* we decided not to separate the mobile object from the RSR functionality. In Section 3.3 we elaborate on the advantages of this approach in detail.

Ease-of-use We believe that ease-of-use is one of the crucial design considerations, as it affects the time required to develop an application and the costs of its maintenance. As observed in [19], there are users for which "a shorter learning curve, ease of program design, development and debugging are just as important as speedups". Based on our experience, this is very true for the parallel mesh generation community.

Following are the aspects which we consider important in achieving ease-of-use of a runtime system:

- flexibility provided to the application, appropriateness of the model;
- clear and simple runtime system API (usability);
- variety of platforms where the runtime system can be used (portability);

- interoperability of the runtime system with the existing software (openness).

Based on our experience with AMR applications, the one-sided communication model together with the data mobility functionality has a number of advantages. This model better addresses the needs of the targeted applications. Because of this, it requires less time and expertise from the application developer. The best possible application performance can be achieved using TCP, MPI or some similar low-level library directly. However, based on our experience, the required developer expertise, development and especially maintenance costs make this approach unacceptable.

In *Clam* we attempt to refine the API of the DMCS/MOL implementation. We do this by merging the functionality of the two layers and by customizing the API based on our previous experience with ILB (both DMCS and MOL were developed years before ILB, which currently is the the main "user" of *Clam*). At the same time, the *Clam* API is sufficient for implementing explicit load balancing techniques directly. While not particularly interesting from the outside of PREMA, the API refinement is very useful for the system maintenance and for the ILB development.

The runtime system portability defines the difficulty of adapting to the changes in the runtime system environment. As far as an application is concerned, most of the systems issues are hidden within the runtime system. In order to be portable, the runtime system should not be tightly dependent on operating system (OS), low-level communication substrate, compiler etc. We view the low-level communication substrate portability as the most important portability feature.

It is appealing to use MPI as the universal communication platform for the runtime system: MPI implementations are available on most OSs and support most of the interconnects; there are numerous groups, which work constantly on improving the quality of the industrial-strength implementations, adding new features and providing user support. While DMCS was supposed to be portable, the actual implementation was coupled with MPI. Nevertheless, only a tiny subset of the functionality provided by MPI was required (namely, initialization and point-to-point communication). The difficulties with porting DMCS on MPICH-G2, which we described earlier, convinced us that the implementation should not depend on MPI.

There is another reason not to be fully dependent on MPI. The requirement of interoperability, or openness [19], defines that it is desirable for an application programmer to be able to use lower-level communication primitives. AMR is usually a component of a composite end-to-end application. The lack of interoperability from the AMR part can affect the rest of the components within such application. This is possible if the runtime system is implemented on top of MPI, while both the runtime system and MPI are used by the application. Such scenario can lead to a deadlock if there is pending MPI communication within the runtime system [15].

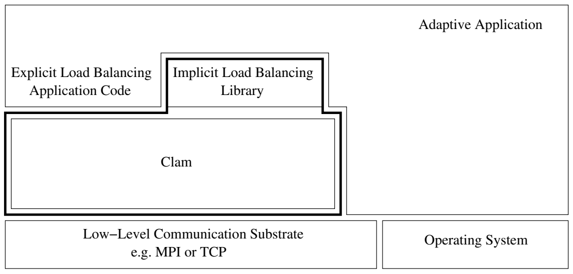
Adaptive Application
Explicit Load Balancing Application Code
Implicit Load Balancing Library
Clam
Low–Level Communication Substrate e.g. MPI or TCP
Operating System

Figure 2. The positioning of *Clam* within PREMA.

Finally, the use of MPI as the only communication platform of *Clam* would restrict its potential capabilities. In particular, to the best of our knowledge, the existing implementations of MPI use a single channel for communication between two processors. In PREMA, it may be desirable to have multiple communication channels, as motivated in [15].

3.3. Implementation

We have outlined the most important design considerations we wanted to incorporate into the implementation of *Clam*. Next, we describe how to meet these design requirements, the costs associated with it, and how the new implementation compares to DMCS/MOL.

The issue of correctness remains the same in *Clam* as it was in DMCS/MOL. However, we identified a problem with the initial implementation of the PREMA communication layer. Inside DMCS, all of the communication requests to the runtime system translate into one or two *send*s, which are matched by *recv*s on the destination processor. Specifically, the first message carries information about the message type. It is followed by the buffer in case of RSRN or message request with a non-zero buffer. In DMCS a blocking *MPI_Recv()* is used to receive the buffer. This approach is deadlock-prone [20]. In the absence of applications at the time when DMCS was developed this problem was not noticed. In *Clam* all requests to the low-level communication subsystem are non-blocking. The improved correctness comes at the price of additional data structures and decreased performance.

We address the issue of performance and ease of use by merging the functionality of DMCS and MOL in *Clam*. The previously taken approach implemented the mobile object functionality on top of DMCS. This makes perfect sense from the software engineering perspective. Unfortunately, this decision led to the (1) duplication of functionality of DMCS within the MOL API, thus duplication of the internal data structures which serve the same function in DMCS and MOL; (2) additional memory copies; (3) use of *both* DMCS and MOL by ILB to improve performance. Based on our experience with DMCS/MOL, there are too few benefits and too many concerns about the effectiveness of this separation to justify it. By merging the functionality of DMCS and MOL we eliminate all of the aforementioned problems and significantly reduce the size of the code to maintain.

The ease-of-use and portability were the important priorities set for *Clam*. The initial motivation argument behind the development of *Clam* was to implement a system not dependent on MPI. In order to do this, we define an Abstract Communication Interface (ACI) within *Clam*. ACI is a small module which implements the necessary communication primitives. All of the low-level details are hidden within the ACI implementation. The *Clam* operations which require communication result in posting asynchronous requests to the ACI module. We have developed TCP and MPI implementations of ACI. The TCP ACI implementation still uses MPI for the startup. We do not consider it worthwhile to re-implement the startup and execution control functionality provided by LAM MPI, although we may have to do this for the Globus port of *Clam* in the future.

Clam is implemented in C (while DMCS/MOL in C++) to improve the portability of the system. The C++ portability issues arise because of the differences in C++ compilers and STL implementations across different OSs. While these difficulties can usually be resolved, C implementation can eliminate them completely. The use of C++ in DMCS/MOL was primarily motivated by convenient implementations of the STL data structures. In *Clam* we use efficient data structures derived from the Linux kernel [10].

As a side-effect of the redesign decisions we have made, the total code size was reduced from around 6000 lines of DMCS/MOL to roughly 4000 of the *Clam* implementation (yet, *Clam* provides some of the functionality, not present in DMCS/MOL). The size of the static libraries compiled with the Sun Workshop `CC` compiler with `-g` flag reduced from 6.8 Mb of DMCS/MOL to roughly 0.6 Mb of *Clam* (mostly because we do not use STL). These is a quantifiable evidence of the improved maintainability (code size) and usability (binaries size) of the runtime system.

We also addressed maintainability by implementing a memory manager within *Clam*. This simplifies debugging and improves memory usage patterns (we implemented a restricted version of slab caching [9] within the *Clam* memory manager).

Clam implements six Location Management Policies (LMPs) as a part of the mobile object functionality. An LMP provides location-independent routing of messages to mobile objects. It was shown in [15] that there is no single LMP which can give the best application performance for all classes of applications. Compared with DMCS/MOL, *Clam* provides the choice of LMP to applications.

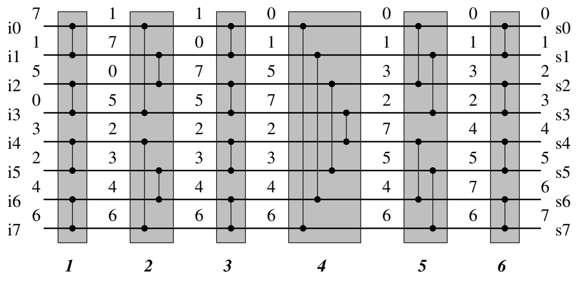

Figure 3. Sorting network for eight inputs.

4. Experimental Results

The experimental study was performed on the SciClone Cluster of The College of William and Mary. We used Typhoon (64 Sun Ultra 5, single UltraSPARC IIi 333/2MB, 256MB mem) and Whirlwind (64 Sun Fire V120, single UltraSPARC IIe 650/512KB, 1GB mem) Solaris 9 subclusters. The nodes within subclusters are connected with 100Mbps FastEthernet. As the MPI implementation we used LAM MPI 7.0 with `usysv` RPI.

4.1. Benchmarks

Synthetic microbenchmark Parallel network sort benchmark, which we call `netsort` for historical reasons, implements a bitonic sorting network. A sorting network is a comparison network which specifies a sequence of comparisons for its input to produce a sorted sequence [13]. The process of sorting a sequence of eight numbers using a sorting network is illustrated in Figure 3. Through the series of comparisons and exchanges, the input sequence *i* transforms into the sorted sequence *s*. For the input 0 this results in the sequence of comparisons with inputs 1, 3, 1, 7, 2 and 1. Each of the shaded regions corresponds to a stage. All the comparisons within each stage can be done concurrently. The reader is referred to [15] for the detailed description of the benchmark implementation.

We used two versions of the `netsort` benchmark. In `netsortC` all of the mobile objects are created on the same processor, while in `netsortD` each processor creates an equal share of the mobile objects. The explanation of the benchmark behavior can be found in [15]. For the evaluation purposes, we are interested only in the relative performance of the two implementations we compare.

It is important to note that the `netsort` benchmark was developed with the purpose of simulating a communication intensive tightly coupled application, like the Optimistic Delaunay mesh generation method [17]. The benchmark was not designed to achieve high performance and speedups of parallel sorting. `netsort` does not use ILB; it is built directly upon the mobile object functionality of *Clam* or MOL.

Parallel Constrained Delaunay Triangulation (PCDT) PCDT is a parallel mesh generation algorithm based on Delaunay triangulation [18]. The reader is referred to [11] for the detailed description of the algorithm and for the definitions of the related terms.

The main difference of the PCDT algorithm from Delaunay triangulation is that the *point cavity* cannot expand beyond the predefined boundary. At the preprocessing stage of PCDT, the problem is divided into a number of subdomains satisfying certain boundary conditions. The subdomains can then be triangulated almost independently on separate processors. The process of subdomain triangulation consists of selecting and changing "bad" triangles (i.e., those, which do not satisfy certain geometric requirements) from the initial triangulation. The recalculation of the subdomain mesh can lead to modifications of the edges located on the subdomain boundary. When a new point has to be inserted on the boundary, a *split* message is sent to the neighboring subdomain located on a remote processor.

The implementation of PCDT decomposes the initial domain and distributes resulting subdomains (as PREMA mobile objects) among the processors, which mesh the subdomains concurrently. The ILB layer of PREMA provides dynamic diffusion load balancing for PCDT. The coverage of the load balancing algorithms implemented in ILB can be found in [5–7].

4.2. Performance Evaluation

We conducted a series of tests both to compare *Clam* with the DMCS/MOL implementation and to measure the absolute overheads of *Clam*. The first test measures the maximum achieved bandwidth over the 100 Mbps link using the ping-pong method. In this test *Clam* RSRN functionality is used to invoke a remote function with a buffer of variable size as an argument. The bandwidth for *Clam* is measured for both the TCP and MPI ACI implementations. The same test test is done for DMCS, MOL and pure MPI. The results for small and large message sizes are shown in Figure 4.

The results of the ping-pong test show that in most of the cases *Clam* achieves better bandwidth than DMCS and MOL. For small message sizes, LAM MPI performs best. For large messages *Clam* implemented on top of TCP ACI gives better performance than pure MPI. This is explained by the three-way handshake protocol used by LAM MPI for messages larger than 64Kb. The performance gain of *Clam* over the similar functionality in MOL is over 20%. The ping-pong data also shows that the overhead MOL adds over the functionality of DMCS (the sources of this overhead were described earlier). Both the MPI and TCP implementations of *Clam* outperform DMCS.

The second test was designed to evaluate the performance of the mobile message functionality of *Clam* and MOL. An object is created on processor 0, migrated to processor 1, and a mobile message is sent to that object from processor 0. When the message is received, a *reply* RSR is invoked on processor 0 from the processor where the object is located. The latency measured in this test is defined as the time between sending a message to the mobile object and receiving the reply. Next, the object migrates to processor 2, and the procedure is repeated. Lazy Forwarding LMP [15] is used both in *Clam* and MOL for this test, so when the object is located on processor 2, *each* message sent to it from 0 traverses through processor 1 to 2 (no location updates; message forwarding is common for appli-

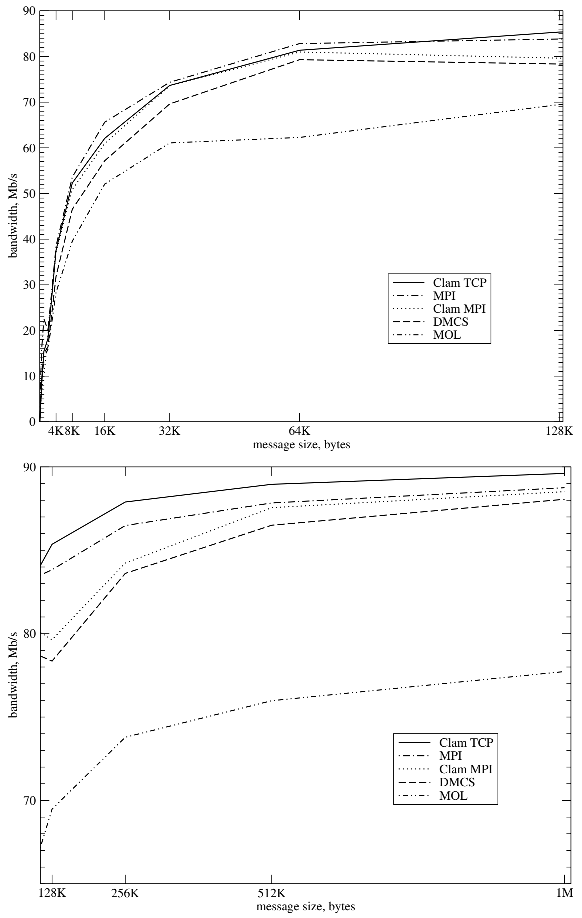

Figure 4. Maximum achieved bandwidth for small and large message sizes.

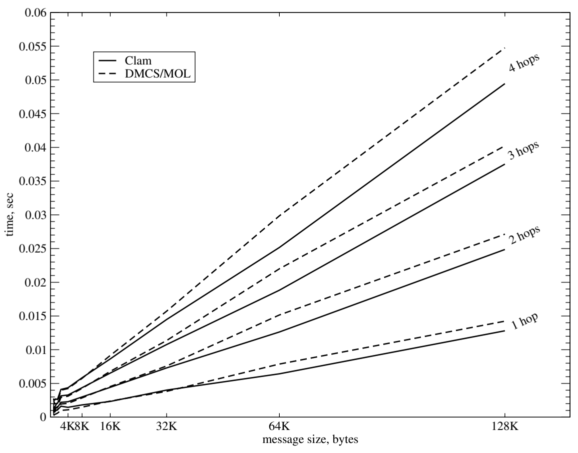

Figure 5. Mobile object message latency test.

cations with frequent object migrations). The results of the test are presented in Figure 5 (the object sequentially migrates from processor 0 to 1, 2, 3 and 4). MOL has lower latency only for small messages as it uses blocking (unsafe) *MPI_Recv* operation. Because of the structural deficiencies, the DMCS/MOL latency over *Clam* slightly increases with the number of hops.

The last two performance tests evaluate the overall effectiveness of *Clam*. Figure 6 plots the runtimes of the `netsort` benchmarks implemented with *Clam* and MOL (both *Clam* and MOL use the same Jump Update LMP [15] for this test). The benchmarks use the messaging functionality of the runtime system extensively. The performance of these operations is better in *Clam*, as shown in the previous experiment. The data also show that *Clam* is more scalable. The results of the `netsort` test validate our decision to merge the functionality of DMCS and MOL. The significant difference in performance of `netsortC` and `netsortD` is explained by the centralized/decentralized creation of mobile objects [15].

Finally, Figure 7 gives the runtime breakdown for PCDT which is using PREMA ILB functionality implemented on top of *Clam* (in this test we had 512 subdomains of the 2-D pipe cut model; the algorithm generated about 35 million triangles; the subdomains were assigned area bounds between 1.92e-2 and 0.26e-2). The overhead introduced by *Clam* is within 5% of the execution time, while the total time was reduced by over 35% compared with the non-balanced version.

5. Discussion and Future Work

We described the runtime system which provides communication runtime support for dynamic load balancing of irregular adaptive applications, like mesh generation. We outlined the design and implementation decisions and the lessons learned which we consider important for any similar runtime system. We presented a qualitative and quantitative evaluation and comparison of the runtime system with the DMCS/MOL implementation previously used in PREMA. Our comparison shows better performance, correctness guarantees, and improved maintainability of the runtime system.

We learned from our experience with DMCS/MOL that it is very difficult to foresee all of the problems, design a good API and correctly balance the design priorities in the absence of applications. This was the case at the time DMCS was developed in mid-90s, when we had no AMR software implemented. We learned a lot from the applications developed recently and improved the runtime support based on that experience.

A number of ideas still remain to be realized within *Clam*. We have to consider the multi-threaded implementation of the communication component and extend the range of the ACI implementations. The experience with ILB showed that it is necessary to be able to receive load balancing messages concurrently with the application computations [6]. Currently this is achieved by periodic polling in a separate load balancer thread. The utility messages are distinguished from the application messages using *tags*. However, this turned out to be an unsatisfactory solution for applications which generate high network traffic [15]. Congestion of the single processor-to-processor communication channel leads to late arrival of the load balancer messages and thus outdated, ineffective load balancing decisions. We plan to address this issue by providing an API (within *Clam*) for creation of additional communication networks (not possible in MPI ACI, but can be done with TCP), similarly to communication bundles in AM [16].

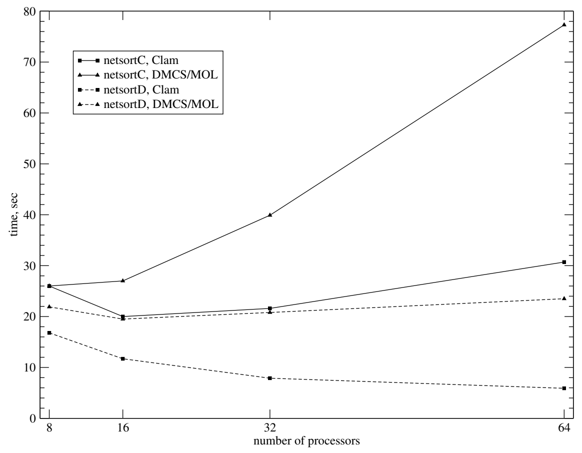

Figure 6. Execution times of the `netsort` **benchmark with DMCS/MOL and** ***Clam*****.**

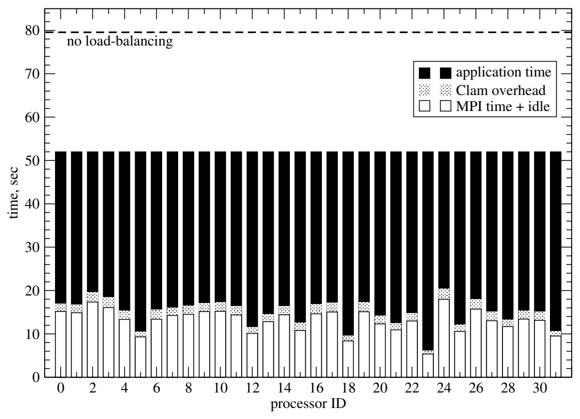

Figure 7. PCDT runtime breakdown (PREMA diffusion load balancing).

This may help to eliminate the channel congestion problem and the existing requirement for message tags.

While *Clam* is still under development, it is an open project and can be obtained for research purposes by contacting the authors.

Acknowledgments

The initial version of the `netsort` benchmark was designed and implemented by Chris Hawblitzel. The PCDT application was developed by Andrey Chernikov. Andriy Fedorov thanks Kevin Barker for valuable discussions about the DMCS/MOL implementation.

This work was performed using computational facilities at the College of William and Mary which were enabled by grants from Sun Microsystems, the National Science Foundation, and Virginia's Commonwealth Technology Research Fund.

The authors are grateful to anonymous referees for the constructive comments on the initial version of this paper.

References

[1] The Globus Project. <`http://www.globus.org/`> (06/10/2003).

[2] MPICH-G2 Mailing List Archives, April 2004. <`http://www-unix.globus.org/mail_archive/mpich-g/2004/04/msg00001.html`> (05/12/2004).

[3] MPICH-G2 Mailing List Archives, August 2002. <`http://www-unix.globus.org/mail_archive/mpich-g/2002/08/msg00009.html`> (05/12/2004).

[4] MPICH-G2 Mailing List Archives, May 2003. <`http://www-unix.globus.org/mail_archive/mpich-g/2003/05/msg00026.html`> (05/12/2004).

[5] K. Barker. *Runtime Support for Load Balancing of Parallel Adaptive and Irregular Applications*. PhD thesis, Department of Computer Science, The College of William and Mary, 2004.

[6] K. Barker and N. Chrisochoides. An Evaluation of a Framework for the Dynamic Load-Balancing of Highly Adaptive and Irregular Parallel Applications. In *Proceedings of SuperComputing'03*, 2003.

[7] K. Barker, N. Chrisochoides, A. Chernikov, and K. Pingali. A Load Balancing Framework for Adaptive and Asynchronous Applications. *IEEE Transactions on Parallel and Distributed Computing*, 15(2):183–192, 2004.

[8] K. Barker, N. Chrisochoides, J. Dobellaere, D. Nave, and K. Pingali. Data Movement and Control Substrate for Parallel Adaptive Applications. *Concurrency: Practice and Experience*, 14(2):77–101, 2002.

[9] J. Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer*, pages 87–98, 1994.

[10] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2nd edition, 2003.

[11] P. Chew, N. Chrisochoides, and F. Sukup. Parallel Constrained Delaunay Meshing. In *Proceedings of 1997 Joint ASME/ASCE/SES Summer Meeting, Special Symposium on Trends in Unstructured Mesh Generation*, 1997.

[12] N. Chrisochoides, K. Barker, D. Nave, and C. Hawblitzel. Mobile Object Layer: A Runtime Substrate for Parallel Adaptive and Irregular Computations. *Advances in Engineering Software*, 31(8–9):621–637, 2000.

[13] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[14] E. W. Dijkstra. Shmuel Safra's version of termination detection, 1987. Manuscript EWD998-7, <`http://www.cs.utexas.edu/users/EWD/ewd09xx/EWD998.PDF`> (11/02/2003).

[15] A. Fedorov. *Location Management in a Mobile Object Runtime Environment*. MS thesis, Department of Computer Science, The College of William and Mary, 2003.

[16] A. Mainwaring and D. Culler. Active Message Applications Programming Interface and Communication Subsystem Organization. Technical Report Draft, University of California at Berkeley. <`http://now.cs.berkeley.edu/AM/am-spec-2.0.ps`> (02/11/2004).

[17] D. Nave, N. Chrisochoides, and P. Chew. Guaranteed-Quality Parallel Delaunay Refi nement for Restricted Polyhedral Domains. In *Proceedings of the 18th Annual ACM Symposium on Computational Geometry*, pages 135–144, 2002.

[18] J. R. Shewchuk. Tetrahedral Mesh Generation by Delaunay Refi nement. In *Proceedings of the 14th Annual ACM Symposium on Computational Geometry*, pages 86–95, 1998.

[19] A. Singh, J. Schaeffer, and D. Szafron. Experience with Parallel Programming Using Code Templates. *Concurrency: Practice and Experience*, 10(2):91–120, 1998.

[20] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *19th Annual Symposium on Computer Architecture*, 1992.