SAND REPORT

SAND2006-0421
Unlimited Release
Printed January 2006

BEC: A Virtual Shared Memory Parallel Programming Environment

Jonathan L. Brown, Sue Goudy, Mike Heroux, Shan Shan Huang, Zhaofang Wen

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Sandia National Laboratories

SAND2006-0421
Unlimited Release
Printed January 2006

BEC: A Virtual Shared Memory Parallel Programming Environment

Jonathan L. Brown, *Sue Goudy[†] Mike Heroux[‡]
Shan Shan Huang [§] Zhaofang Wen[¶]

Abstract

We propose an abstraction, named BEC, to enable Global Address Space (GAS) capabilities for parallel programming in SPMD style. It is a portable lightweight approach for incremental acceptance of the GAS model, along an evolution path that leverages existing infrastructures and maintains backward compatibility with existing programming methods and environments. It assists migration of legacy applications thereby encouraging their expert programmers to adopt the new model.

In addition, it provides for some of the unaddressed needs, such as efficient support for high-volume fine-grained and random communications, which are common in parallel graph algorithms, sparse matrix operations, and large scale simulations. The idea behind BEC is that messages are aggregated by a runtime library for bulk transport to handle such unpredictable communication patterns. Data from initial experiments with a prototype communication bundling library using the Bundle-Exchange-Compute (thus motivating the name BEC) programming style shows that this approach scales well. As examples of suitable BEC applications, we present sparse matrix kernels for multiplication and overlapping Schwarz preconditioning [5, 11]. We also discuss solid mechanics material contact [1, 18] with abundant irregular, fine-grained communication.

BEC can be used as an enhancement to existing environments such as MPI. It can also function as an intermediate language [14] to other high level GAS languages such as PRAM C [8] and UPC [30]. Furthermore, it can serve as a bridge between programming models such as virtual shared memory and message passing.

*Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI 48109
†Sandia National Laboratories, Albuquerque, NM 87185, email: spgoudy@sandia.gov
‡Sandia National Laboratories, Albuquerque, NM 87185, email: maherou@sandia.gov
§College of Computing, Georgia Institute of Technology, Atlanta, GA 30332
¶Sandia National Laboratories, Albuquerque, NM 87185, email: zwen@sandia.gov

Acknowledgment

We would like to express our appreciation to Bill Camp for his constructive criticism, encouragement, and support for our programming model research. We would like to thank Neil Pundit for his helpful coaching, and Jim Ang for his enthusiastic support for this research. Many thanks to David Womble for providing student funding through CSRI. Ron Brightwell and Bruce Hendrickson pointed out the link between the BSP model and BEC. Thanks to Ron Brightwell and Rolf Riesen for helpful discussions on OS, MPI, and network issues. Mike Glass, Kevin Brown, and Courtenay Vaughan helped us a great deal by providing us with valuable insight into internal workings of Sandia software for simulation of material contacts. Bruce Hendrickson and Steve Plimpton taught us about algorithms used in Sandia's contact libraries. Doug Doerfler and Brice Fisher asked pertinent questions which motivated us to formalize the BEC model. This research used resources of the National Energy Research Scientific Computing Center at Lawrence Berkeley National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

We wish to acknowledge distinguished leaders for their contributions in the field of Global Address Space programming model research and for inspiring us. These notably include Bill Carlson, Kathy Yelick, and Bob Numrich. In particular we gratefully acknowledge Bill Carlson's guidance and his assimilation of Zhaofang Wen and Mike Heroux into the UPC Consortium. Thanks also to the UPC Consortium and, in particular, Lauren Smith, Tarek El-Ghazawi, Phil Merkey, Steve Seidel, and Dan Bonachea.

Special thanks to Lewis Goudy for proofreading the document, which greatly enhanced the presentation.

Contents

1 Introduction 7
  1.1 Our Approach 8
  1.2 Role of Programming Models 9
  1.3 Historic Perspective 10
2 BEC Model 11
  2.1 Overview 11
  2.2 Language Extension 12
  2.3 Execution Model 14
  2.4 Support Infrastructure 14
3 BEC in Context: Related Models 17
  3.1 Fine-Grained Access to Shared Data 18
  3.2 PRAM C 18
  3.3 Comparison of PRAM C and UPC 19
  3.4 BEC's Relationship to MPI and High Level GAS Languages 20
  3.5 Relationship to the Bulk-Synchronous Parallel (BSP) Model 20
4 Performance 22
  4.1 Linked List Ranking 22
  4.2 BEC Prototype Results 23
  4.3 UPC Implementation 23
  4.4 Analysis 30
5 Sparse Matrix Examples 31
  5.1 Sparse Matrix-Vector Multiplication 32
  5.2 Overlapping Schwarz Preconditioning 33
  5.3 Using BEC for Row-based Matrix-Vector Multiplication 34
6 Conceptual BEC Approach for a Large-Scale Application 35
  6.1 Finite Element Method with Material Contacts 36
  6.2 Global Address Space Concepts for Contact Simulation 38
  6.3 Discussion of Potential BEC Approach to Contact Simulation 40
7 Conclusion 43

Appendix

A BEC Language Specification 47
  A.1 Types 47
  A.2 Language Extension 47
  A.3 BEC Interface Object 49
B Code Examples: Parallel Linked List Ranking 51
  B.1 Linked List Ranking in BEC 51
  B.2 Compiler Generated C Code 54
C Prototype Code Overview 57
  C.1 Library Classes 57
  C.2 Parallel Ranking Code 58

Figures

1 BEC Supporting Infrastructure. 16
2 Execution time for BEC prototype, lists of 128 to 8K items. 23
3 Execution time for BEC prototype, lists of 16K to 256K items. 24
4 Execution time for BEC prototype, lists of 512K to 4M items. 25
5 Relative scaling for BEC prototype, lists of 16K to 1M items. 26
6 Execution time of UPC parallel ranking code. 27
7 Execution time of UPC parallel ranking code by node count. 28
8 Relative scaling of UPC parallel ranking code by node count. 29
9 Comparison of BEC prototype and UPC execution time. 31
10 Partitioned matrix A. 32
11 Partitioned vectors x and y. 32
12 Overlapped matrix A on (a) first processor and (b) second processor. 34
13 Random communication in a parallel contact example. 37
14 Finite element partition centric algorithm for contact enforcement. 41
15 Global contact list centric algorithm for contact enforcement. 42

BEC: A Virtual Shared Memory Parallel Programming Environment

1 Introduction

GAS stands for Global Address Space. On distributed memory hardware, GAS provides ease of programming by allowing one processor direct access to another processor's memory. GAS promises improved productivity over existing programming environment (such as MPI); and therefore it is attractive for high productivity computing systems (HPCS).

The DARPA HPCS program is entering its phase III (2006-2010) with a focus on programming models [15]. GAS is regarded as a key step toward HPCS goals. Major parallel machine vendors have dedicated teams developing their own future models, all to offer GAS as a subset. GAS languages such as Co-array Fortran (CAF) and Unified Parallel C (UPC) have received endorsement by universities, industry, and government agencies. CAF and UPC are already available on many computing platforms. Selected applications written in CAF and UPC have demonstrated comparable performance with MPI; and the GAS versions required only a small fraction of the development time compared to the MPI versions [23, 25]. However, there are still significant challenges on the path to true adoption of GAS models by the parallel programming community.

A practical programming model must balance the often conflicting technical factors such as application performance, expressiveness and ease of use (for programmer productivity), scalability to increasing number of processors, and portability to a wide range of architectures.

Furthermore, there are non-technical barriers to acceptance of new programming models. Legacy applications represent significant past investment. Expert application programmers of legacy models are comfortable with existing environments. Migration of legacy applications and their expert programmers, along with the cost of development and adoption of a new model, pose real challenges to any new programming model effort.

There are two distinct paths for introduction of new programming models: build everything from scratch while ignoring legacy applications and users; and introduce new programming models gradually through a bootstrapping process that leverages existing infrastructures.

1.1 Our Approach

We choose an incremental path to introduction of new programming models. BEC is an abstraction that enables GAS capabilities for parallel programming in SPMD style. It is a lightweight approach for the gradual adoption of the GAS programming model. In addition, it provides for some of the unaddressed needs, such as efficient support for high-volume fined-grained and random communications, which are common in parallel graph algorithms, sparse matrix operations, and large scale simulations. As examples of suitable BEC applications, we present sparse matrix kernels for multiplication and overlapping Schwarz preconditioning [5, 11]. We also discuss solid mechanics material contact [1, 18] with abundant irregular, fine-grained communication.

In contrast to other existing GAS languages, some of which build everything from scratch, the BEC approach is an inexpensive and portable way to introduce GAS capabilities. It is minimally intrusive and compatible with existing programming methods.

The BEC abstraction can be implemented as a minor language extension to high-level languages such as C and Fortran, plus a thin runtime layer which serves as an interface to the underlying communication infrastructure. Since this runtime layer is so thin, its bulk-transport function can be manually fine-tuned to take advantage of the communication capabilities of the platform.

One copy of the thin runtime layer runs on each processor. It bundles remote communication requests into message queues according to their destination physical processors. The underlying communication infrastructure is then called upon to exchange the remote requests. After that, remote requests from other physical processors are resolved; and results are sent back. The returned remote data is maintained locally by the runtime layer for future references by the user program. (Note that these activities require little user intervention.)

As detailed in this report, an implementation of the BEC abstraction focuses on a C language extension plus this thin runtime layer. (For convenience, this implementation is also referred to as BEC.) Specifically, the C extension includes a *shared* data type and a very simple API for the runtime layer. In a program, accesses to a shared data structure must first be "requested" through the API to the BEC runtime, which bundles up the requests for remote data. By a simple call to the API in a subsequent programming step, the request bundles collected by the BEC runtime, are exchanged among the physical processors to resolve all the requests. After the exchange, computation involving shared accesses can proceed.

The idea behind BEC is motivated in part by dynamic communication bundling techniques proposed in [8]. In that fundamental approach messages are aggregated automatically by a runtime library for bulk transport to provide efficient support for fine-grained communication that may be both irregular and random. Such communication patterns are common in parallel graph algorithms, sparse matrix operations, and many large-scale simulations. Data from initial experiments with a prototype communication bundling library using the

Bundle-Exchange-Compute (thus motivating the name BEC) programming style shows that this approach scales well; and it outperforms the the current UPC implementation [30], which does not scale on the test problem: a parallel linked list ranking [22] algorithm that exhibits high-volume fine-grained random shared memory accesses.

Another motivation of the BEC idea comes from the fact that MPI programmers often develop ad hoc message queues and communication utilities to dynamically bundle up messages in order to reduce total communication overhead. The introduction of BEC will formalize and simplify the creation and usage of message bundling. In implementation, the BEC runtime layer can leverage some existing communication utilities, such as the EPETRA package in Trilinos [20].

BEC can be used as an enhancement to an existing environment such as MPI. It can also function as an intermediate language [14] to other high level GAS languages such as PRAM C [8] and UPC [30]. Furthermore, it can serve as a bridge between programming models such as virtual shared memory and message passing.

1.2 Role of Programming Models

Parallel programming models provide an abstraction for programmers to develop good parallel software. A model provides mechanisms for the movement of data between processors and for the synchronization of processors. A good model abstracts away unnecessary hardware details, while, at the same time, it encourages a programming style designed to exploit the underlying system architecture to yield high performance. Parallel programming models have yet to achieve the success of completely abstracting hardware details while maintaining good performance, as has been done for sequential computation, and thus the two most common parallel programming models mirror the most common system architectures: shared memory and message passing.

Shared memory programming provides the user with globally addressable memory space. This is usually implemented in hardware, and suffers from architectural scaling limits. Shared memory programming usually uses barrier constructs for synchronization, as there is no implicit synchronization with a read or write to a shared location. Shared memory programming has no explicit communication, and thus it becomes difficult to pinpoint where in user code improvements could be made to reduce communication and improve performance. Further, seeking only to minimize accesses to shared memory locations does not address the actions of cache-coherent hardware in modern shared memory systems, which can generate data movement for cache updates or invalidations. However, shared memory programming more closely approximates the flexibility of Parallel Random Access Machine (PRAM) programming [34] than does message passing, and as such can be considered a more natural model for expressing parallel algorithms.

By contrast, message passing is an architecturally driven programming model, with hardware considerations driving the programming style. Commodity or near-commodity processors can be connected with message passing networks. Message passing code has explicit read and write calls, which also provide for synchronization, and the very successful message passing library, MPI, includes a vast array of common communication collective operations. Since communication is explicit, it is easy to pinpoint where communication occurs in message passing code, and thus alter the user code to reduce communication.

1.3 Historic Perspective

For decades researchers and language developers have been exploring and proposing parallel library and language (PLL) extensions to support large-scale parallel computing. In this entire time, MPI has been the only project that can be called a broad success. PVM [13] and SHMEM [24] have made an impact on a subset of platforms and applications. Co-Array FORTRAN [31] and UPC [30] are still active efforts with some promise of being widely adopted. Shared memory parallel models such as POSIX [35] Threads and OpenMP [33] are also extremely useful, but large-scale parallelism using threads is limited by a number of factors such as a lack of computers with large processor counts, problems with latency and data locality of logically shared data that is physically distributed and subtle issues such as false cache line sharing that can make a parallel program much slower than its serial counterpart.

Ironically, the success of MPI has made the adoption of true language extensions, and other novel library approaches extremely difficult across existing parallel application bases and within existing parallel application development teams because there is a high degree of satisfaction with the performance and availability of MPI and a critical mass of MPI expertise. In other words, many people think MPI is all they need. Many good ideas have failed because they have not recognized and addressed this attitude. There are still many opportunities to improve upon MPI, both in usability and performance. In particular, we can improve upon MPI in the following ways:

- Readability: Because MPI is a set of library interfaces, MPI code tends to be less readable than a language extension such as the square bracket notation of Co-Array FORTRAN or the SHARED keyword of UPC. Language extensions tend to make code easier to develop, understand and maintain.
- Runtime Overhead: Although MPI function call overhead can be negligible for coarse-grained parallelism, it can be significant for very fine-grained parallelism. Language extensions can avoid this overhead.
- Compiler Optimization: MPI function calls are opaque to the compiler. Therefore, remote memory references, which in principle could be scheduled and buffered, are not visible to an optimizing compiler.
- Specialized interfaces: MPI is a general-purpose low-level library. This is excellent for flexibility, but makes it difficult to implement specialized communication patterns

because they must be built up as an unwieldy sequence of low-level calls. Language extensions support irregular remote memory access in a natural way with loop expressions using indexed reads and writes to get or put remote data, without recourse to buffering or registering user data types to a library. This capability is arguably the most important practical feature of Co-Array Fortran and UPC.

An additional problem in the area of language extensions is the "chicken-and-egg" dynamic. Any language extension will not be used by applications unless compilers will support it, and compilers will not support it unless applications are obviously willing to use it. So a carefully orchestrated bootstrapping process must be managed to reach a critical mass of use and support. However, Sandia is an excellent environment for this kind of bootstrapping process since we have research and development efforts throughout the entire vertical spectrum from hardware, OS and programming environments to new application development.

From our perspective this is the challenging environment any PLL project needs to address. The efforts described in this report are intended to provide a suitable alternative or enhancement to MPI, capabilities that MPI cannot provide, or both.

2 BEC Model

BEC can be viewed as an enhancement to existing message-passing systems, such as MPI. It provides a virtual shared memory interface with explicit software control for data communication. BEC is also designed to overcome one of the shortcomings of shared memory programming in that it allows programmers to identify when communication occurs. In this section, we present an overview of programming in BEC, the BEC language extension and compiler support, BEC execution model, and the support infrastructure for BEC. For additional details, see Appendix A.

2.1 Overview

BEC is a formalization of the Bundle-Exchange-Compute programming style. BEC presents its users with convenient language extensions (to ANSI C) and library calls. Its execution is supported by the compiler and runtime. In addition to BEC's language extensions and library, programmers can also use MPI and C in their BEC programs.

BEC follows the Single Program Multiple Data (SPMD) model. There is an instance of the BEC runtime object executing on each physical processor. BEC programs should roughly adhere to the following structure:

- Issue requests for reads of shared variables.

- Issue exchange call. This call signals the runtime to fetch the shared values requested. Exchange calls also write through any outstanding write requests, which are implicitly generated by assigning value to a shared variable.
- Local computation resumes, with the requested values available for a read by the requesting processor. For convenience, we refer to the instructions executed between consecutive exchange calls as executing within the same *phase*, with exchange calls separating distinct phases of execution.
- Optional barrier calls to synchronize amongst processors. The Exchange call is also a barrier, but with data exchange.

Note: All reads must be explicitly requested. Multiple uses of the same data only need to be requested once for one exchange call. The user can request an array section (say elements $B[20:50]$). Writes need not be requested explicitly; they will be registered at runtime by the BEC runtime library. Physically remote (shared) data will only be available after the exchange call, which carries out the actual transporting and processing of remote communication requests. Remotely fetched values will be stored in a local software cache, which is a component of the BEC runtime library.

Next, we present the language extensions of BEC and the compiler support, showing how each of the above steps are accomplished.

2.2 Language Extension

BEC adds the following extensions to ANSI C:

- `shared`: The `shared` keyword, when used to modify a C variable declaration, creates a shared region. If this is outside any function, exactly one shared region will be created. If this is in a function body, one region will be created per execution of the declaration. If the type of the declaration is modified by the star, the shared region will be of pointer type to a shared region of the base type. If the `shared` keyword appears in the formal arguments of a function, then the shared variable will be passed by *reference*, when the function is called.
- `BEC_joint_allocate`: This is a space allocation function. It is called by all processors to create exactly one shared region. There is no guarantee of synchronization.
- `BEC_local_allocate`: This is another space allocation function. It is called by only one physical processor to create a shared region. The creation notice is sent to other physical processors on the subsequent exchange call.
- `BEC_request`: Shared data *must* first be requested with the BEC_request call to make them available in the next phase of computation. `BEC_request` takes as parameters a

shared variable, an offset, size of data needed, and an optional "persistence bit" which, if set to 1, will keep the region available until explicitly released. If no persistence bit is set, the shared region will only be available until the next exchange call unless requested again. The size parameter allows programmers to fetch data of arbitrary length, which is often useful in fetching a segment of a shared array.

- `BEC_exchange`: `BEC_exchange` causes data communication. Read requests are served, and writes are recorded. The exchange call takes an optional parameter to clear all persistence bits.

- `BEC_barrier`: `BEC_barrier` is a wrapper on the underlying transport layer's barrier call. It can be used as a synchronization tool. `BEC_exchange()` acts as a barrier also, but should be called when data exchange is needed.

- `PROCOF`: A function that takes a shared variable as a parameter, and returns the process ID of the physical processor owning the byte identified by the argument.

- `MY_PID`: A variable that contains the physical processor ID of the calling processor.

In addition to extensions listed above, BEC also provides functions to manipulate persistence bits and to check whether specific shared locations are available to read. BEC can also expose runtime functions for advanced tuning.

2.2.1 Compiler

The BEC compiler has the following functions:

- Inserting `Initialize()` and `Finalize()` calls to appropriate places in the program to initialize and clean up Shared Memory Manager.

- Rewriting shared variable declarations to the declaration of `shared_ref_t` type.

- Calling the appropriate Shared Memory Manager allocate functions, `BEC_JointAlloc` or `BEC_LocalAlloc` for shared variables that have static memory storage requirements (i.e., `shared int a[5]`). For shared pointer declarations where the programmer does not immediately allocate space, (for example, `shared int* a[5]; ... a[0] = BEC_JointAlloc (...)`), the compiler generates calls to allocate space for the `shared_ref_t` types created by these later allocation calls, and write the later-created `shared_ref_t` to the shared region.

- Generating read/write calls to the Shared Memory Manager when shared variables are referenced, as well as creating local variables to hold the values of shared memory reads/writes. For example, indexing into a shared variable is an implicit read/write, which can be translated to a `BEC_read` or a `BEC_write` call, before computation can continue. The following is an example translation of an addition/assignment involving shared arrays A, B, and C. (Note: This translation scheme is provided for illustration purposes; many optimizations are possible to improve such a simple scheme.)

```
A[i] = B[i] + C[i];
```

Translation:

```
tempB = BEC_read(B, i, ...)
tempC = BEC_read(C, i, ...)
tempA = tempB + tempC;
BEC_write(A, i, &tempA, ...);
```

- Translating PROCOF and MY_PID references to the appropriate runtime calls.

2.3 Execution Model

A translated BEC program will execute on a user-determined number of processors. A BEC program follows the SPMD model: programs are executed asynchronously except when explicit synchronization is requested (i.e., via BEC_barrier, or BEC_exchange functions).

Shared memory is managed by the Shared Memory Manager (Section 2.4.1), which is part of the BEC runtime library. The Shared Memory Manager interface provides functions to read from and write to shared memory regions. These functions use MPI as the underlying communication library. Before any access to shared memory, a call to function Initialize() is made. This function creates an instance of the Shared Memory Manager for the calling processor. After the final access to shared memory, a call to function Finalize() is made to clean up local memory taken up by the Shared Memory Manager. Both of these calls are generated by the BEC compiler.

BEC expects users to explicitly request the exchange of data. This means that before any BEC_exchange call is issued, any writes to shared memory will not be seen by other read requests. If a read follows a write to the same location in the same phase, the result may be undetermined.

2.4 Support Infrastructure

BEC is designed to be interoperable with existing message passing libraries. Its support infrastructure is designed in layers and implemented in C++:

- **Bulk Transport**
  At the bottom level, the Bulk Transport is a wrapper on the underlying message

passing library. It is designed to move large arrays between processors in a synchronous fashion. The Bulk Transport is designed to shield the upper-level objects from changes in underlying message-passing libraries, as well as to allow for performance tuning.

- **Bundler**
  Above the Bulk Transport, the Bundler implements a push/pull interface with mailboxes for all other processors. The Bundler builds aggregate messages on-the-fly, in response to pushes of small messages, and, on an exchange, passes transfer buffers to the Bulk Transport for communication. After an exchange, upper-level objects can access messages received via the pull call.

- **Shared Memory Manager**
  The top level of the infrastructure stack, the Shared Memory Manager maintains the virtual shared memory components, and converts reads and writes to the appropriate action that is passed to the Bundler.

- **BEC Object**
  The BEC object is a shallow object that implements an interface that acts as a wrapper on the Shared Memory Manager and other support objects and libraries. It provides a consistent interface for the translator/compiler.

The infrastructure supporting BEC is depicted in Figure 1.

2.4.1 Shared Memory Manager

The Shared Memory Manager is responsible for maintaining all data structures to support BEC virtual shared memory. It maintains a map of shared references to Partition Objects and Range Query Objects, described in subsequent sections. This permits identification of owning processors for shared regions, as well as the maintenance of a software-managed cache of remote values in a searchable data structure. The Shared Memory Manager implements request, read, write, and exchange function calls, which are exposed upward to the BEC language extension.

2.4.2 Partition Objects

Shared regions, though globally addressable, exist as distinct memory regions on each of the physical processors. Partition Objects are responsible for maintaining the map from global logical offsets to physical offsets on specific processors. Partition Objects implement functions to identify the owning processor of subregions of a shared memory region, as well as to find physical addresses on the local processor for subregions located on that processor.

The type of Partition Object is selected at allocation. Because allocation can be done by a single processor, there is an option to serialize a Partition Object to produce a configuration

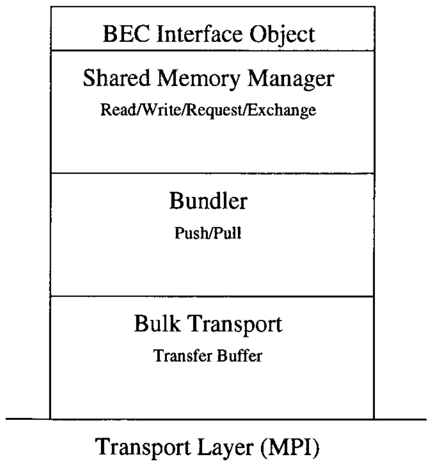

Figure 1. BEC supporting infrastructure. BEC is designed to work with existing message-passing transport layers, such as MPI. The Bulk Transport is responsible for transferring large buffers between processors. The Bundler is responsible for receiving small messages, aggregating these, then separating received aggregate messages after an exchange call. The Shared Memory Manager implements the virtual shared memory interface. The BEC Interface Object provides a thin wrapper for the support infrastructure.

string, and to read that configuration string into a Partition Object. A Partition Object Factory is responsible for creation of Partition Objects upon allocation using a convenient index for choosing the type of Partition Object.

2.4.3 Range Query Objects

The BEC model uses a structured bundle-exchange-communicate programming pattern. Data is requested, requests are bundled, then data becomes available after the exchange operation for the next communication phase. This data needs to be stored until the next exchange operation. Further, if a shared memory location is marked as persistent, it is updated on every exchange, and also must be stored. The Range Query Object is responsible for storing and managing these local copies of shared data.

The Range Query Object gets its name from its interface: it implements a function for locating the local pointers to ranges of bytes in a shared region. One Range Query Object exists for each shared region on each processor.

3 BEC in Context: Related Models

A comprehensive survey of past and present parallel programming model research can be found in [6]. This section only covers closely related models.

GAS models can be realized in libraries and language extensions. Examples of GAS libraries are Cray's SHMEM [24], and MPI-2 [32]. Existing library-based GAS models support global address space programming in the sense that one processor can access another processor's memory without the remote processor's cooperation. GAS language extensions based on similar ideas include Unified Parallel C (UPC), Co-Array Fortran (CAF) [31], and Titanium (Java-based) [12]. These languages typically support global address space through virtual shared memory, such as UPC "shared array" and co-array in CAF. Virtual shared memory is physically distributed when the language is implemented on a distributed machine. Virtual shared memory provides convenience in programming because the shared memory can be accessed at random by all the processors regardless of the actual physical data layout.

In addition to the shared data type, GAS languages also provide language constructs to express parallelism. For example, UPC provides a parallel loop construct as follows.

```
upc_forall(i = 0; i < N; i++; A[i]) {
    A[i] = B[i] + C[i];
}
```

The UPC_forall loop looks like the C for_loop except for the addition of the fourth expression in the loop header. This additional expression is called the *affinity expression* as a tip to the compiler. In this example, the compiler will try to assign i-th iteration of the loop to the physical processor that hosts shared array element $A[i]$.

3.1 Fine-Grained Access to Shared Data

The convenience in virtual shared memory programming can sometimes lead to (intentional or unintentional) abuse in random fine-grained accesses to shared array elements. Access to shared array elements in other processors requires communication, which is more expensive than access to local data. Too much fine-grained communication can cause significant performance penalty because of the communication overhead (latency) in each separate transaction.

To avoid (or minimize) fine-grained remote accesses, several conditions are necessary.

- The problem solution does not require random fine-grained accesses, but this is not true for graph algorithms, sparse matrix manipulation, and many large scale applications (such as Sandia simulations that have extensive material contacts).
- Furthermore, the fine-grained access pattern in a problem solution needs to be regular to match the limited and strict data layout patterns of the GAS language (such as those specified by block size in UPC shared array declarations). Programmers need to be careful about data locality in shared data structures in order to avoid or minimize fine-grained remote accesses. (To solve the data locality problem, an application support layer [7] is proposed to manage data locality for UPC applications. This layer includes domain-specific libraries which "map" the natural array access patterns in algorithms to the physical data layout patterns of UPC. Such a support layer would require substantial time and resources to develop and mature.)

3.2 PRAM C

Recently, another GAS language, named PRAM C, has been proposed [8] to address the performance penalty for fine-grained access to (physically remote) shared data structures. PRAM C is a simple extension of C. Like other GAS languages, PRAM C supports shared array as well as a couple of language constructs to express parallelism. For example, PRAM C has the following construct.

```
PRAM_do(N): f(...);
```

where $f()$ is a function whose instances would be executed in parallel by N virtual processors. The PRAM_do construct is introduced to match the semantics of the theoretical

PRAM model (Parallel Random Access Machine [27]). PRAM was the model of choice for two decades of parallel algorithm research [22]; and volumes of published parallel algorithms are available for implementation.

The PRAM C approach to fine-grained communication is not to avoid it, but to reduce the total overhead of all (fine-grained) communication in an application. This approach depends on two key ideas working together: automatic communication bundling by the runtime library and other opportunities to bundle communication owing to the PRAM semantics.

3.3 Comparison of PRAM C and UPC

For comparison we focus on the main control constructs of the two languages, PRAM_do and upc_forall. Both constructs are used to express parallelism, up to N parallel threads of execution. In PRAM_do, N instances of function $f(...)$ will be executed in parallel; while in upc_forall, N instances of *body* will be executed, possibly in parallel.

But there are many fundamental differences.

Nested upc_forall constructs are reduced to sequential for-loops. Nesting of PRAM_do is allowed and the parallel semantics of the nested PRAM_do is unchanged.

There is also some difference in semantics. Statements in $f(...)$ of PRAM_do follow implicit barriers ([8]); so the parallel execution of the instances of $f(...)$ is synchronous. Statements in a upc_forall *body* follow only explicit barriers ([29]), so the parallel execution of the instances of *body* is asynchronous.

PRAM_do enables expression of parallelism in virtual processors. That is, each instance of the function $f(...)$ will be executed by one virtual processor. In upc_forall, each instance of *body* somewhat resembles an instance of function $f(...)$ in PRAM_do. But they are different.

- In PRAM_do, each instance of $f(...)$ can have its own local variables, which are truly private to the executing virtual processors. (Communication between different instances of $f(...)$ can only be through shared variables or shared arguments.)
- In UPC local variables pertain to physical processors. Therefore, local variables in UPC are potentially shared by multiple instances of the executing *body* assigned to run on the hosting physical processor. So local variables in UPC are not private to one instance of the body, but are really shared by a group of instances. In this sense, upc_forall does not fully support expression of parallelism in virtual processors.

There is also difference in how work is assigned to physical processors.

- Depending on the affinity expression in upc_forall, the group of instances assigned to a physical processor needs to be determined at compile time (and even at runtime).

Consequently, users need to make sure that sharing of local variables by multiple *body* instances does not lead to unexpected program behavior. Explicit synchronizations may be needed for program correctness; but such synchronization points apply to all the instances on all physical processors. Currently, there is no synchronization mechanism in UPC to deal with such sharing of local variables on each physical processor.

- PRAM_do does not have such an issue since all the local variables are truly private to the virtual processors. Work assignment to physical processor can either be done at compile time or runtime, beyond the concern of the user.

3.4 BEC's Relationship to MPI and High Level GAS Languages

BEC can serve as a bridge between MPI and high level languages such as PRAM C. BEC can be used as an enhancement to MPI; and it runs on top of and together with MPI. Also, a PRAM C program can be translated into a BEC program [14].

Like PRAM C, BEC provides virtual shared memory for ease of programming, but without any of the automatic data movement or synchronization. It relies on explicit request and global exchange steps to make remote data available, similar to a message passing system. BEC emphasizes bundling fine-grained communication into larger packets, then calling the runtime library in explicit steps to exchange and process these packets. It is well designed for the massively thread multiplexing envisioned for PRAM C, offering fine-grained access to the programmer while using the underlying message passing network to transfer larger, aggregate packets.

3.5 Relationship to the Bulk-Synchronous Parallel (BSP) Model

BSP is an abstract model proposed by Valiant [36]. It is defined as the combination of three attributes:

1. A number of components, each performing processing and/or memory functions;

2. A router that delivers point-to-point messages between pairs of components; and

3. Facilities for synchronizing all or a subset of the components at regular intervals of L time units where L is the periodicity parameter. A computation consists of a sequence of supersteps. In each superstep, each component is allocated a task consisting of some combination of local computation steps, message transmissions and (implicitly) message arrivals from other components. After each period of L time units, a global check is made to determine whether the superstep has been completed by all the components. If it has, then the machine proceeds to the next superstep. Otherwise, the next period of L units is allocated to the unfinished superstep.

3.5.1 BSPlib

Much research has been done on the BSP model since its first introduction [21]. One significant example is BSPlib, developed as an alternative to MPI[16] and PVM [13]. BSPlib is a runtime library that provides support for the BSP style parallel programming. It is already being used by a rapidly growing community of software developers in universities and in industry. BSPlib can be used with C, C++, or Fortran. It supports SPMD parallelism based on efficient one-sided communications. The core library (excluding collective communications) consists of just 20 primitives. The Oxford BSP toolset includes profiling tools and implementations of BSPlib for many different machines [21].

3.5.2 BEC vs. BSP

There is similarity between the BEC model and the theoretical BSP model. That is, computation and communication in BEC are performed in phases clearly separated by the *BEC_exchange*() call, which serves as a barrier. This is similar to "superstep" in the BSP model, with a minor difference that in BEC there is no need for parameter L to control the regular time interval for a superstep. As to the similarity, BEC can be seen as a practical implementation of the abstract BSP model.

However, BEC provides shared data type as virtual shared memory to support Global Address Space programming, which is not addressed in either the abstract BSP model or BSPlib. For this reason, BEC supports a higher level of abstraction (thus improved ease of programming) than BSP/BSPlib. Therefore, BEC can be seen as a BSP implementation plus virtual shared memory.

Another important aspect of BEC is that it can serve as a GAS intermediate language to other higher level GAS languages such as UPC [30] and PRAM C [8], which can be compiled into BEC [14]. The theoretical significance is that this is the first time that fine-grained parallel PRAM programs (in PRAM C) can be compiled into coarse-grained BSP style programs (in BEC).

Extending BEC futher, so as to support asynchronous programming, may increase the difference between BEC and BSP.

4 Performance

Performance numbers were measured on seaborg, an IBM SP2 system at NERSC. Seaborg is configured with shared memory nodes, having sixteen processors each, connected by a high speed network. MPI can be used within or between shared memory nodes. The BEC prototype code used MPI calls exclusively for communication between processors, whether on the same node or between nodes.

4.1 Linked List Ranking

Given a linked list, the problem of list ranking is to compute the distance from each node to the tail of the list. This problem is of interest to us as a test case for the effectiveness of BEC on code exhibiting fine-grained, random, irregular communication. Linked list ranking is a fundamental operation in parallel algorithm design, described in many parallel programming texts and tutorials (e.g. [22]). We consider a list of n items, where each item i has a pointer $p(i)$ to its successor in the list. The final element in the list has a special nil pointer. At the end of the algorithm, we would like each item i $(0 \leq i \leq n-1)$ in the list to hold an integer $r(i)$ that signifies its distance to the tail of the list, and thus is its rank in the list. To compute the ranking in parallel, p and r are realized as shared arrays.

We implemented a straightforward algorithm to solve the linked list ranking problem for a list of n items: We create three shared arrays jp, p, and r, each of n elements. We assume that the pointer array, p is initialized to the linked list values. At the outset, each item i has a "jump pointer" $jp[i]$, which is initialized to $p[i]$. We initialize $r[i]$ to 1. We loop $\lceil \log n \rceil$ times. In each iteration j, each item i adds $r[jp[i]]$ to $r[i]$, and sets $jp[i]$ to $jp[jp[i]]$. An item stops participating once $jp[i]$ is nil.

This algorithm basically does pointer jumping in parallel. In this context, pointer jumping refers to indirect addressing of the elements of the global shared array. Here indirect addressing uses the value of an array element as an pointer (index) to get to another array element. Since the pointer values are random, this algorithm produces many fine-grained messages that are unpredictable and irregular. And therefore parallel linked list ranking provides a good example of the type of algorithm that does not perform well in a straightforward MPI implementation but may benefit from the BEC communication bundling and aggregation strategy. The above algorithm, when executed by n PRAM processors, is not work optimal, as it takes time $O(\log n)$. The usual solution is to simulate the n PRAM processors on a smaller number of processors, but, as our goal is to explore the performance of this algorithm, not necessarily to provide the optimal solution to this problem, we eschew the added complexity. For our code, we use a globally addressable BEC shared array to store arrays p, jp, and r. Each of P MPI processes are then responsible for n/P list items.

Lists were generated in a quasi-random fashion with a balanced tree library and the standard `rand()` system call, seeded by the time of execution. Lists were generated on one

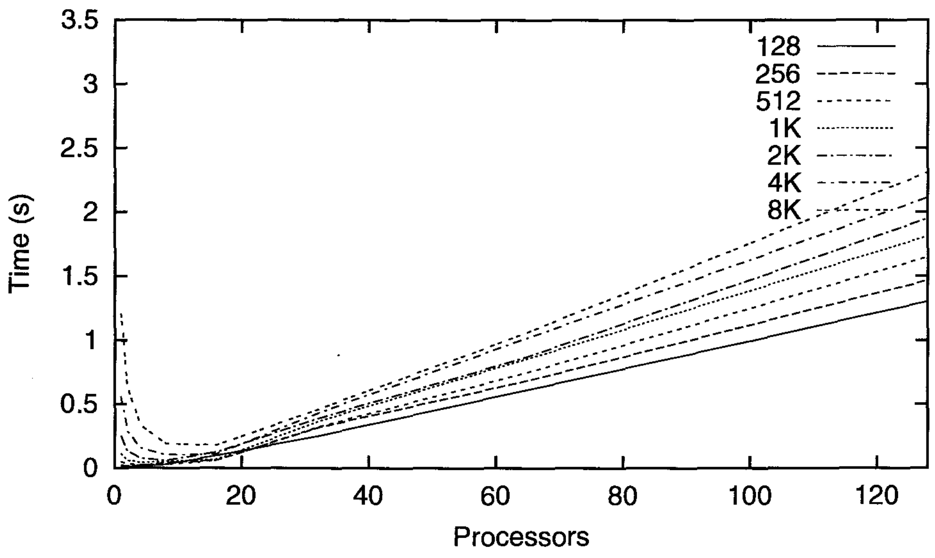

Figure 2. Execution time for BEC prototype, lists of 128 to 8K items. With small lists, the overhead of the prototype implementation is clear. We see a fixed time cost of 0.1 to 0.3 seconds, and execution time scales up with both increasing processor count and increasing list size.

processor and distributed with an MPI scatter call. To simplify the sequential implementation, we assume that the list starts with element 0.

Parallel execution was timed from before the scatter call to the end of the corresponding gather call. For details about the BEC prototype implementation, please see Appendix C.

4.2 BEC Prototype Results

The BEC prototype was tested on lists of size 128 to $4M$ (2^{22}) items, and on $P = 1$ to 128 processors. The execution time is shown in Figures 2, 3, and 4. We present scaling relative to a single processor's execution of the parallel code in Figure 5. We observe that, once the list became sufficiently large, reasonable relative scaling was observed.

4.3 UPC Implementation

A UPC version of the parallel linked list ranking code was also developed and benchmarked. Due to limitations on the shared segment size and time constraints, the maximal list size used was $1M$ (2^{20}) items. This was sufficient to see several general trends in the results.

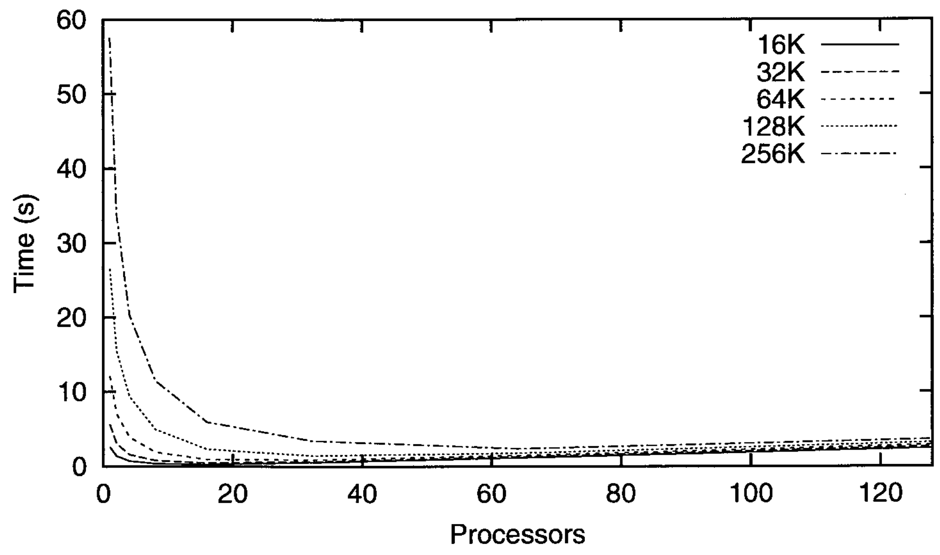

Figure 3. Execution time for BEC prototype, lists of 16K to 256K items. With these moderately-sized lists, we see an initial trend of falling execution time for increasing processor counts. However, we observe diminishing returns for 64 and 128 processors, as the execution time curves level-off at 32, then start to rise up to 64 and 128 processors. This is evidence of a cost trade-off between computation per processor and communication cost of adding additional processors; with these list sizes, dividing only the work more than offsets the added communication cost for up to 32 processors.

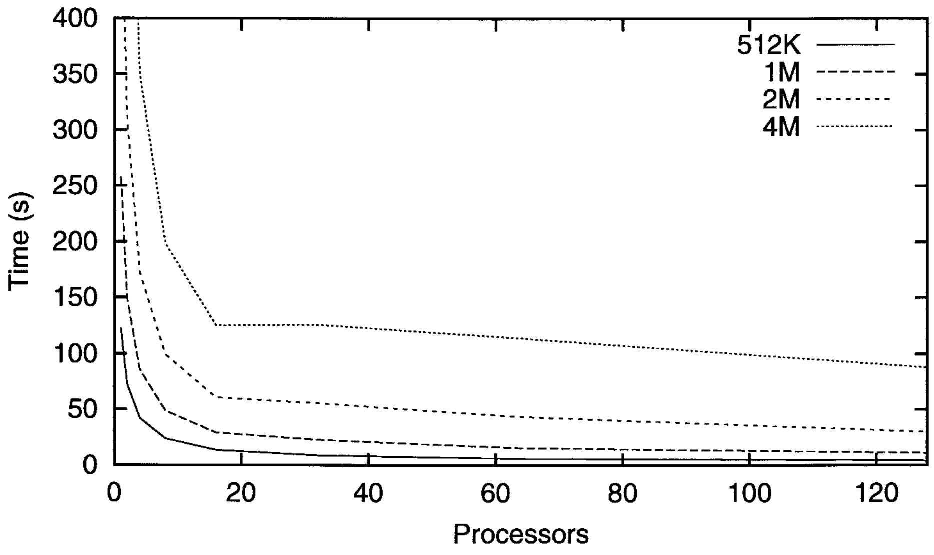

Figure 4. Execution time for BEC prototype, lists of 512K to 4M items. With these larger lists, we see a continual advantage for adding processors. The execution time appears to be free of L1 cache effects, as the L1 cache is now far too small for the working set, and the 8 MB L2 cache available on the test system is sufficient for the working set.

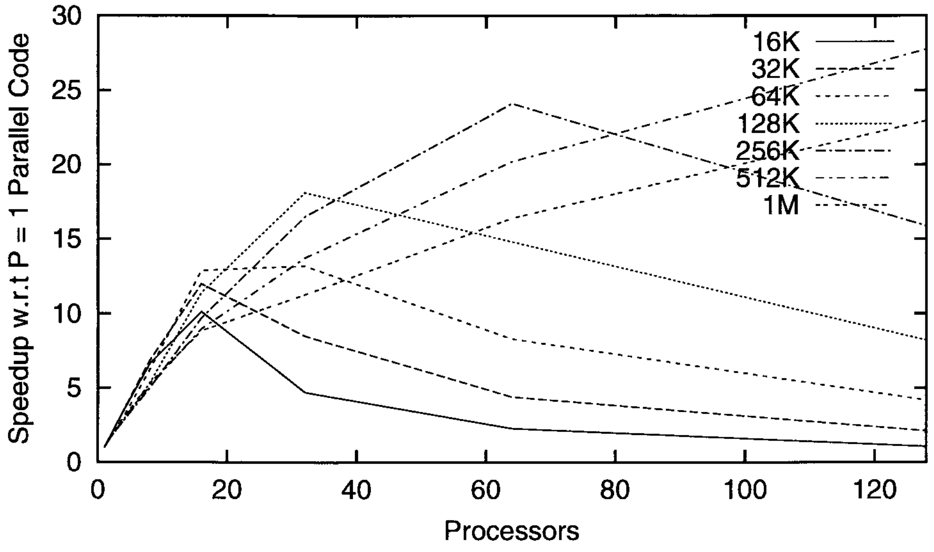

Figure 5. Relative scaling for BEC prototype, lists of 16K to 1M items. With increasing list size, we see good scaling up to a ratio of list items to processors of roughly 4096. For example, for input size of $256K$, speedup deteriorates after 64 processors, at an average of 4096 items per processor. This may be an artifact of the implementation – parallel arrays of integers are used to hold the pointers, jump pointers, and counts – as well as cache effects from a 64 KB L1 cache on the test system. We observe that there is a transitional range as this ratio increases up to 16384 items per processor, when the working set would be larger than the L1 cache.

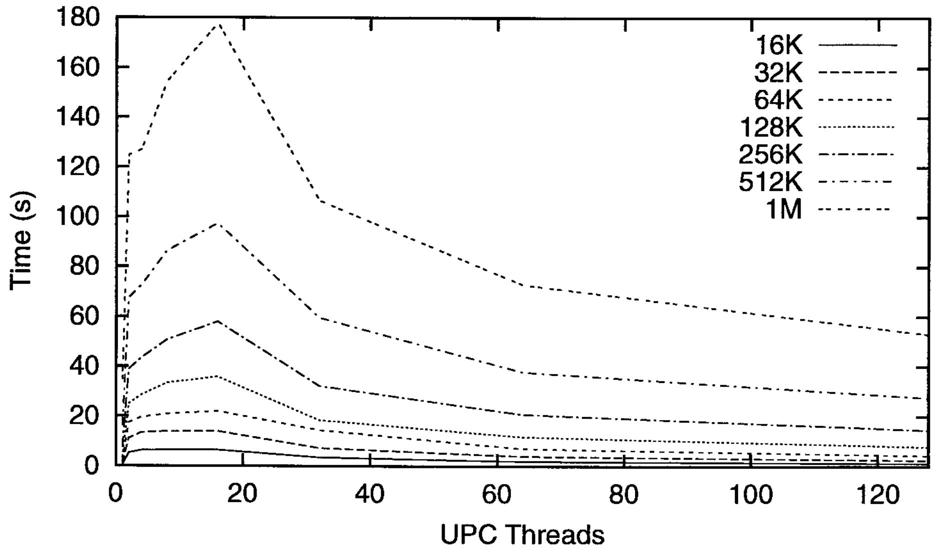

Figure 6. Execution time of UPC parallel ranking code. We observe a trend in execution time of greater cost as the threads per node increases, and less cost as the total number of nodes increases. For example, the execution time uniformly increases up to sixteen threads, but drops at 32 threads, for which two nodes were used. The Berkeley compiler optimizes execution for a single UPC thread, and thus we see dramatically lower execution time for a single UPC thread than any other number. Lists of size $16K$ (2^{14}) to $1M$ (2^{20}) were used due to limitations on the UPC shared region size.

Execution time is presented per UPC thread (Figure 6) and per node (Figure 7). There was a general pattern of execution time increasing on a single node as the number of threads increased on that node, and a trend of decreasing execution time as the number of nodes increased. We provide relative scaling for nodes (Figure 8) to show the scaling with the number of nodes.

Version 2.1.17 of the Berkeley UPC compiler was used, and published performance studies of code generated by the Berkeley compiler use only one processor per shared memory node, thereby forcing communication through the GASNet network layer [10]. It appears that the Berkeley compiler relies on some other mechanism for communication within a shared memory node, and that this mechanism does not perform well. The LAPI library was used, instead of MPI, for better performance on the suggestion of the Berkeley UPC group.

Throughout, one UPC thread was executed per processor. Thus, we may consider "thread" and "processor" to be indistinguishable in this context.

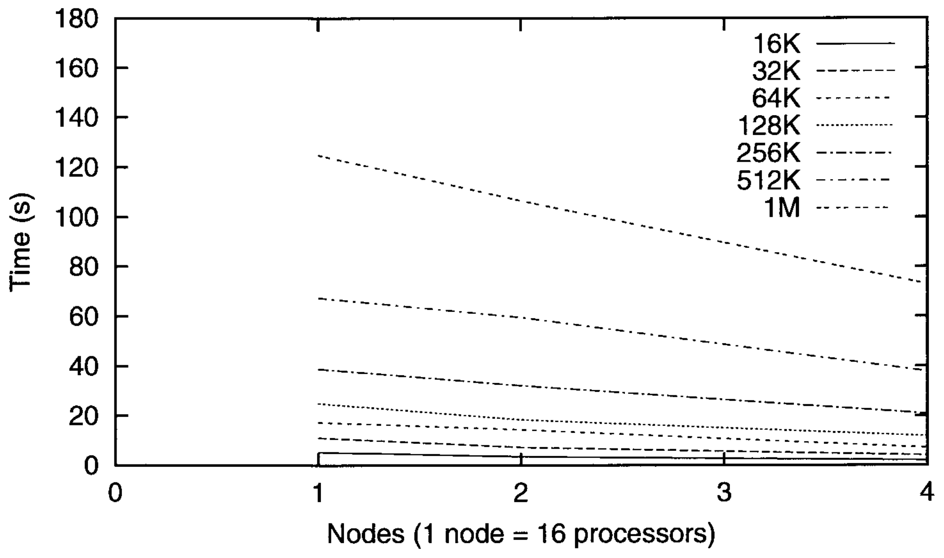

Figure 7. Execution time of UPC parallel ranking code by node count. For one node, two UPC threads per node were used, as this produced the best performance number for a single multiprocessor node that used the UPC runtime. For all others, sixteen threads per node were used. UPC does appear to get a performance gain by using multiple nodes for the parallel ranking code, but the BEC prototype performance remains better in total execution time. Lists of size $16K$ (2^{14}) to $1M$ (2^{20}) were used due to limitations on the UPC shared region size.

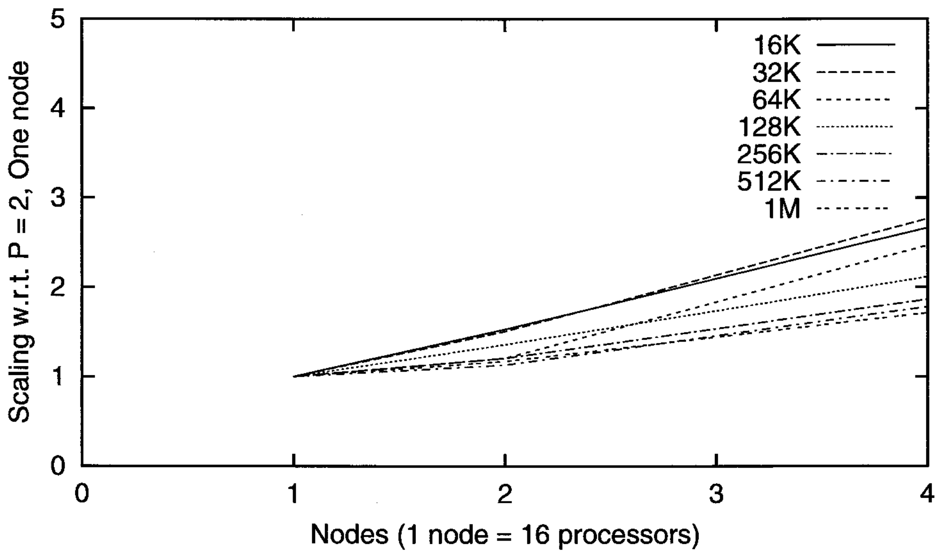

Figure 8. Relative scaling of UPC parallel ranking code by node count. For one node, two UPC threads per node were used. For all others, sixteen threads per node were used. Scaling waned as the list size increased, suggesting that the larger lists were generating communication at a rate $\omega(n)$, and hence we would expect the long-term behavior of UPC to be poor on this problem due to a lack of bundling of fine-grained communication. Lists of size $16K$ (2^{14}) to $1M$ (2^{20}) were used due to limitations on the UPC shared region size.

4.4 Analysis

First, we explain the behavior observed in the execution of the BEC prototype. In general, we have that parallel execution time T is related to T_{comm}, the time of communication, and T_{comp}, the time of computation, by

$$T = T_{\text{comm}} + T_{\text{comp}}$$

The goal of parallel computation is to reduce T by dividing the work across P processors. Here, the parallel execution time behaves as the dominating factor of T_{comp} and T_{comm}. When P increases, T_{comp} decreases (scales) when there are no cache effects; on the other hand, T_{comm} increases. From our experiments, when there are more than $4K$ items per physical processor, T_{comp} dominates; otherwise, T_{comm} does. However, when the number of items per processor is greater than $16K$, the cache effects upset the scaling of T_{comp}.

The division of labor that decreases T_{comp} comes at the cost of added communication T_{comm}, as communication tends to increase both with the volume of data communicated and with the number of processors communicating. Execution time improves if the reduction in T_{comp} is greater than the corresponding increase in T_{comm}. In Figure 2, we see that execution time increases with increasing numbers of processors, indicating that the decrease in computation was insufficient to offset the increase in communication. In Figure 3, we see that the input size grows so that, up to 32 processors, increasing the number of processors is a winning proposition. Increasing beyond 32 processors for lists of $16K$ to $256K$ items provides an insufficient decrease in computation time to offset the rise in communication. Once we reach list of at least $512K$ items, the savings in computation cost is sufficient to achieve continual performance gains for up to 128 processors, as shown in Figure 4. We observe that, in order to maintain positive scaling relative to P, input size per processor needs to be between $4K$ and $16K$ items.

There are two clear conclusions that can be drawn from the data about our BEC prototype and the BEC strategy:

- **Communication bundling can be used to implement a natural algorithm with fine-grained, random, irregular communication on message passing hardware and outperform existing GAS languages.** There is no published source to indicate how the current generation of UPC compilers perform any communication bundling – in fact, in [10], we have that the Berkeley UPC compiler performs no optimizations. Thus, the UPC code is an example of a virtual shared memory programming language permitting fine-grained communication; our BEC prototype outperforms the UPC implementation (Figure 9). Further, our BEC prototype does not display the same scaling pains as input size increases that was observed with the UPC code. For the UPC code, smaller problem sizes showed better performance with greater processor numbers, whereas the BEC prototype continued to do well as the input size increased.

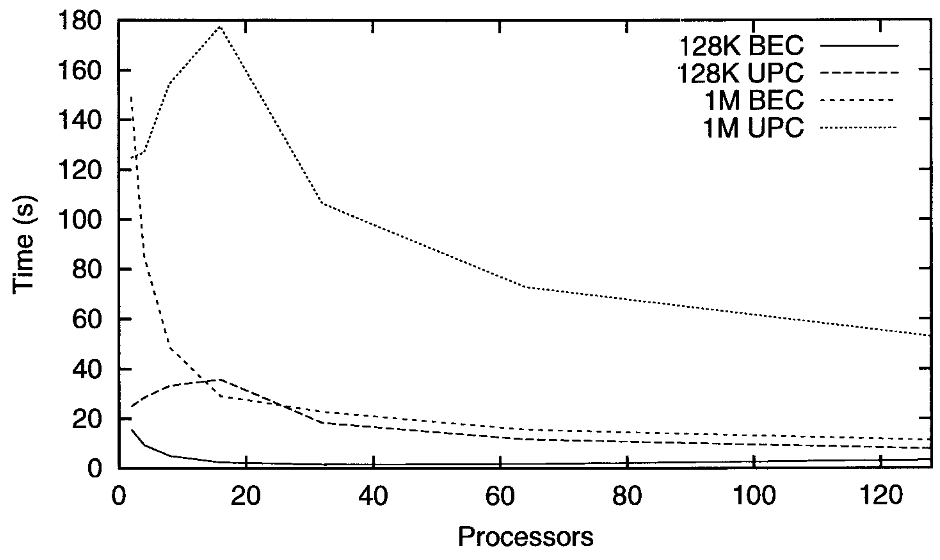

Figure 9. Comparison of BEC prototype and UPC execution time. Time is provided for lists of $128K$ and $1M$ items. Note that, for a single processor, the UPC version will execute as sequential C code, and thus we start at $P = 2$.

- **Additional work is needed to decrease overhead in the BEC support library.** The BEC prototype has several shortcomings:
  - The prototype was not designed for performance, and thus has many memory-to-memory copies. The cost of local memory operations appear to be quite expensive with respect to communication operations – when profiled, the time spent in push/pull operations to pack aggregate packets for transmission was greater than that spent in communicating these aggregate packets.
  - The prototype did not implement the software cache for shared locations, and relied on ad hoc methods for globally indexing into the arrays of pointers and of counts.
  - The use of a circular buffer in software added to the need for memory-to-memory copies, and a better solution should be devised in the next version.

5 Sparse Matrix Examples

Unstructured sparse matrix computations provide several excellent examples that can utilize BEC. To illustrate the types of problems we are addressing and to motivate our design, we begin with a discussion of several examples. Specifically, we discuss:

$$A = \left[\begin{array}{cccc} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ \hline 0 & 0 & a_{33} & a_{34} \\ a_{41} & 0 & a_{43} & a_{44} \end{array}\right] \qquad A = \left[\begin{array}{cc|cc} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ 0 & 0 & a_{33} & a_{34} \\ a_{41} & 0 & a_{43} & a_{44} \end{array}\right]$$

Figure 10. (a) Row-partitioned matrix A (b) Column-partitioned matrix A.

$$x = \left[\begin{array}{c} x_1 \\ x_2 \\ \hline x_3 \\ x_4 \end{array}\right] \qquad y = \left[\begin{array}{c} y_1 \\ y_2 \\ \hline y_3 \\ y_4 \end{array}\right]$$

Figure 11. Partitioned vectors x and y.

1. Sparse matrix-vector multiplication and
2. Overlapping Schwarz preconditioning.

5.1 Sparse Matrix-Vector Multiplication

Sparse matrix-vector multiplication is one of the most important parallel distributed memory kernels for a broad set of applications. For unstructured problems, communication patterns for this kernel are determined at runtime by the nonzero pattern of the sparse matrix. For PDE applications and sparse iterative solver libraries, a row-based partitioning has been commonly used. For example, Trilinos/Epetra [20, 19], PETSc [4, 3, 2] and Aztec [28] all use this distribution by default for general sparse matrices. Figure 10(a) shows this partitioning for a 4-by-4 matrix A on two processors. In this case the first two rows are stored in the local memory of the first processor while the last two rows are stored on the second processor. In contrast, sparse direct solvers are primarily based on column partitioning as shown in Figure 10(b), storing the first two columns of the matrix on the first processor and the last two rows on the second processor.

Regardless of how the matrix is partitioned, vectors are typically partitioned to match the row or column distribution. Figure 11 shows how 4-by-1 vectors x and y would typically be stored to be compatible with A in Figure 10.

5.1.1 Row-oriented multiplication

Assuming our matrix A is partitioned as in Figure 10(a), to compute $y = Ax$ each processor needs to first obtain values of x that are not local. We call this type of data transfer an *import* operation, where we know what we want to receive. For our specific example, the first processor needs x_3 and the second processor needs x_1. Once the off-processor x values are obtained the matrix vector multiplication can proceed without further communication since elements of y are stored on the same processors as rows of A.

5.1.2 Column-oriented multiplication

Assuming our matrix A is partitioned as in Figure 10(b), we can begin computing $y = Ax$ without any communication. However, after this step each processor will have contributions to y that must be sent to the other processor for summation. We call this type of data transfer an *export* operation, where we know what we want to send. For our specific example, the first processor must send its portion of y_4 to the second processor and the second processor must send it portion of y_2 to the first processor. As each processor receives data from other processors it must sum the partial results to obtain the final result for y.

5.2 Overlapping Schwarz Preconditioning

Schwarz methods are a family of domain decomposition methods that introduce coarse-grained parallelism by partitioning the problem domain into subdomains. Assuming one or more subdomains is completely assigned to a processor, on each subdomain the majority of computation can be performed without communication. Overlapping Schwarz methods assign parts of subdomains to more that one processor. Without overlap, preconditioners based on Schwarz methods tend to lose robustness as the number of subdomains increases. With overlap this problem still exists, but is less pronounced. In many practical settings, even though overlap introduces redundant work, there is an overall improvement in performance.

The typical way to implement overlapping Schwarz preconditioners is to create a submatrix for each overlapped subdomain. Given this matrix, we can choose one of many serial preconditioners and apply it to the submatrix as though it were the full matrix. Common submatrix preconditioners are Gauss-Seidel, Incomplete Cholesky or Incomplete LU.

5.2.1 Determining the overlap

Although overlapping Schwarz methods originated in the context of PDEs on continuous domains, it is well known that they can be applied to sparse matrices that are similar in nature to PDEs. In particular, level-based overlap approaches are simple to define. Consider the row-oriented matrix A in Figure 10(a). level 1 overlap is obtained by determining which

$$\text{Overlapped}A_1 = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ 0 & 0 & a_{33} & a_{34} \end{bmatrix} \qquad \text{Overlapped}A_2 = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ 0 & 0 & a_{33} & a_{34} \\ a_{41} & 0 & a_{43} & a_{44} \end{bmatrix}$$

Figure 12. Overlapped matrix A on (a) first processor and (b) second processor.

columns have nonzero entries on a processor and then including the corresponding rows in the overlap. For the first (last) two rows of our matrix A, there is a non-zero entry in column 3 (1). The level 1 overlap matrices are shown in Figure 12.

5.2.2 Factoring the overlapped preconditioner

In order to compute local factors for the overlapped preconditioner we must first import the rows of matrix that are part of the overlap. On each processor, we scan the columns of the owned rows, looking for entries whose column number has no corresponding row number on the processor. The import step then gathers the off-processor rows, typically ignoring any column entries that do not correspond to rows in the overlapped matrix. After the import step, any serial preconditioner can be used, since all data is local and the matrix is square.

5.2.3 Applying the overlapped preconditioner

In order to apply the preconditioner, say $z = M^{-1}r$ as part of a preconditioned iterative method, the same basic import pattern is needed to get off-processor entries of r as we used for creating the overlap matrix. Similarly, we will have extra values of z on each processor that must be resolved. These can be combined as an average using an export pattern that is the reverse of the import, or can simply be set to zero if the entry is does not correspond to an owned row of the original matrix.

5.3 Using BEC for Row-based Matrix-Vector Multiplication

To implement a row-based matrix-vector multiplication algorithm for computing $y = Ax$ using a partitioning as in Figure 10(a) and Figure 11, only a single object needs BEC support, namely the vector x. To improve efficiency, we would first scan the entries of the matrix A on each processor to identify the columns on that processor that have non-zero entries, storing these column numbers in a local array. This would be done as a preprocessing step and is equivalent to knowing which columns are non-trivial on a given processor.

Given this array `column_numbers[]` of length `num_local_columns`, a code fragment that would effectively prefetch the off-processor elements of x is as follows:

```
for (i=0; i<num_local_columns; i++)
  Request(x, column_numbers[i], sizeof(double));
BEC_exchange();
```

At the end of the `BEC_exchange()` call, the appropriate elements of x would be available for computing the local portion of the matrix-vector multiplication kernel. Similar simple constructs would be used for column-based matrix-vector multiplication and for overlapping Schwarz preconditioning, as well as many other distributed unstructured sparse matrix computations.

6 Conceptual BEC Approach for a Large-Scale Application

In this section, we consider the challenges to efficient parallelism that are presented by the simulation of contact. We will focus on solid dynamics; however, the basic steps and algorithms are the same for simulations of quasi-static and transient dynamic solid mechanics. Section 6.1 describes briefly the interactions of material contacts with finite element calculations. In section 6.2 we discuss global address space considerations for detection and enforcement of contact. We propose two algorithms, in the BEC style, for the enforcement phase. Section 6.3 outlines our approach to a BEC implementation of contact enforcement.

From a parallel computation standpoint, it is desirable to consider contact algorithms because these typically exhibit fine-grained and random communication patterns that require a global view. Moreover, the detection and enforcement of contact constraints are important to many Sandia mechanics applications. We point out areas where load balance among the parallel processes may be less than optimal and outline BEC approaches to alleviate this condition.

By its nature, the current implementation of contact simulation is quite complex. The purely computational phases use sophisticated algorithms; the communication between processors requires careful timing and bookkeeping. A measure of this complexity is the number of man-years (approximately 25) that have been invested in the development of the various algorithms, starting with the original serial implementation and continuing to date with the ACME library. We recognize that introduction of new methods into a mature library requires careful analysis and planning. An estimation of the scope of this work is provided in section 6.3.

We gratefully acknowledge conversations with Kevin Brown, Mike Glass, and Steve Plimpton, which helped us simplify the presentation of contact constraint detection and

enforcement. The authors came to understand that enforcement of contact constraints is poorly load balanced in the current contact library. Therefore, we chose to focus on the enforcement computational phase in the following sections, which describe the BEC approach as it could be applied to contact.

6.1 Finite Element Method with Material Contacts

Solid mechanics simulations involve a number of complicated computational phases such as finite element integration, detection of contacts between moving surfaces, and enforcement of contact constraints. Inherent differences in these phases present challenges for efficient parallelization of such simulations. Finite element integration requires local information about geometry of elements and nodes and about physical attributes like location, velocity, material properties, etc. A decomposition based on the finite elements results in effective load balance, with communication mostly between "nearest neighbors" in the collection of processes that are applied to the simulation.

Contact simulations, however, require a global view of the surfaces that can potentially touch each other. Surface interactions result in additional nodal forces in the computation of the finite element mesh. Contact interactions can be extremely complicated in terms of the communications required for detection and enforcement. Figure 13 shows a simple (albeit mechanically unrealistic) example that illustrates how random communication can arise in a parallel contact algorithm.

It is not our intent to mimic the current algorithm, but rather to seek new insights that are possible from a global shared data representation of contact. Therefore, we briefly discuss the current algorithm before turning to global address space considerations. As outlined in [9], the steps for finite element method with contact are these.

1. Compute FE forces on each mesh element.
2. Predict mesh movement without considering contact.
3. Detect mesh/mesh contacts.
4. Generate push-back forces, i.e., enforce contact constraints.
5. Update mesh positions.

A parallel contact detection algorithm was developed from a serial one that is described [17]. In [1], the excellent general description of parallel contact includes the reasons that computations on the finite element mesh are well balanced with respect to workload per processor. The development of a separate decomposition [9, 26] for the geometric surface information results in good load balance for contact detection. In recent years, advances in the calculation of the push-back forces have emphasized the load imbalance of the contact

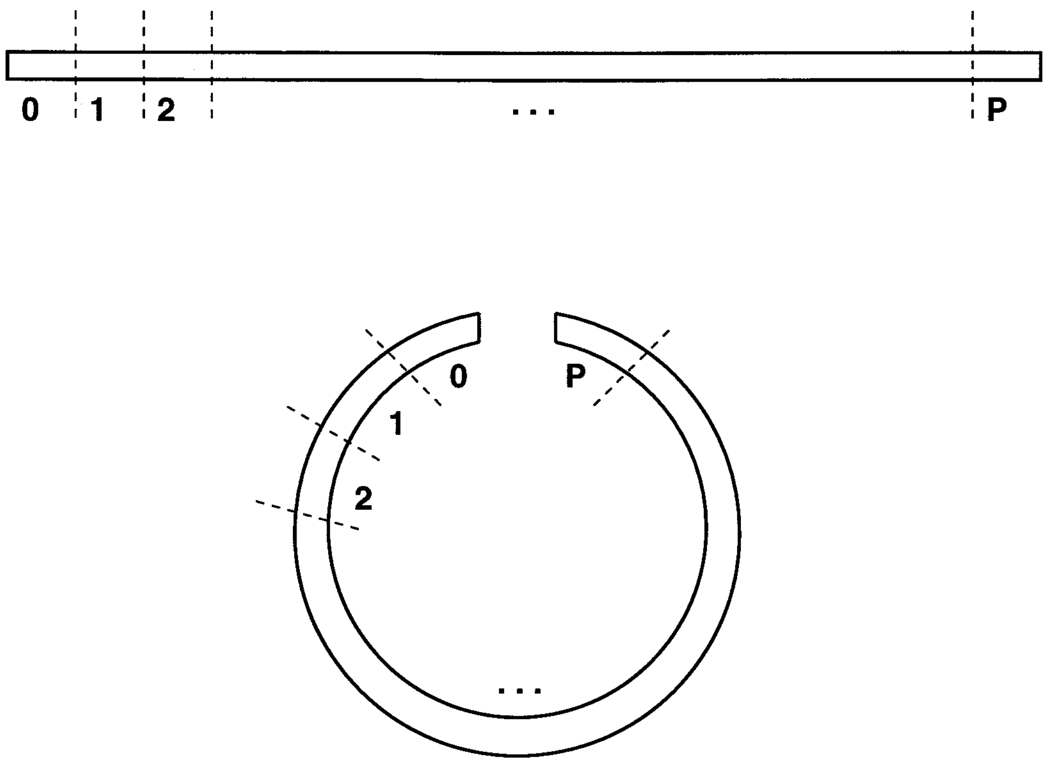

Figure 13. Random communication in a parallel contact example. In the top figure the finite elements in the bar are evenly partitioned across P processors. Force is applied to bend the bar into a circular shape. For some length of time during the simulation, only "nearest neighbor" processes need to communicate. Eventually, the remote processes, logically numbered 0 and P, will hold surfaces that come into contact. The continuing simulation will require that information be exchanged between these two processes. If the simulation continues until the bar is bent into a pretzel shape, then more (and random) processes will need to communicate.

enforcement phase of the simulation. This imbalance arises in part from the use of the finite element decomposition for enforcement due to the need for physical properties that reside in that decomposition. Now let us examine enforcement of contact constraints, taking a BEC point of view.

6.2 Global Address Space Concepts for Contact Simulation

According to [18], the search for contact constraints is global in nature. This attribute makes it natural to approach parallel contact algorithms using a global shared data framework. Because the list of pairs of surfaces that come into contact is dynamic, we choose to think of it as an entity that is formed at the beginning of a simulation and incrementally updated at each time step. This view leads us to reorder the steps of the finite element computation cycle. Moreover, the simple statements about contact list creation and updates are a result of our choice to focus first on contact enforcement. We feel that this focus is reasonable, because the developers who discussed the contact algorithms appeared to be happy with the load balance of parallel contact detection. At a later time, we will examine a BEC global shared approach to contact detection (creation and incremental adjustment of the global contact list). Thus, for our BEC-style algorithms, the general description of a calculation cycle, which emphasizes enforcement, is the following.

Input: shared data structures
FE, containing finite element data, and GCL, global contact list

Output: updated FE and GCL

1. Enforce contact.
2. Move FE mesh.
3. Update contact list.

Before stating algorithms that use a global view of finite element and contact data, a summary of terminology is given here.

- **FE**: finite element data, a *static* shared data structure that contains finite elements with their associated geometric and physical properties
- **FE_partition[i]**: the i-th logical partition of FE, where $0 \leq i \leq P-1$
- **GCL**: global contact list, a *dynamic* shared data structure that contains "pairs of surfaces", which are geometric data describing the surface plus the logical position of the surface in the global FE structure

- **GCL_partition[i]**: the i-th logical partition of GCL, where $0 \leq i \leq P-1$
- **Note**: FE and GCL are shared (global) data structures. The dynamic association of partitions of GCL with processors is not necessarily related to the static association of processors with FE partitions.
- **pair(A,B)**: a pair of contact surfaces denoted by A and B
- **color(surface)**: equals k iff surface belongs to FE_partition[k]
- **color(pair(A,B))**: equals color(B)
- **Note**: $0 \leq \text{color} \leq P-1$
- **fe(surface)**: FE data needed for local contact enforcement calculation
- **$\mathbf{P_i}$**: the i-th logical process of P processes

Now we can present two alternative approaches to contact enforcement simulation, both of which are derived from global views of the finite element and the contact information. Creation of the global contact list involves nontrivial parallel manipulation of geometric data. Both of these algorithmic approaches ignore, for the present time, the details of parallel contact detection.

Algorithm 1 : FE_partition centric approach

1. Group pairs in GCL by color, resulting in color groups CG[i], $0 \leq i \leq P-1$.
2. In parallel, P_i uses CG[i] and FE_partition[i] to do
   (a) contact enforcement (produce new forces);
   (b) finite element calculations (propagate new forces through the mesh);
   (c) produce a new contact sublist CL[i] locally.
3. Assemble the local contact sublists into the global contact list.

Taking a global view allows straightforward derivation of Algorithm 1, which is not too different from the current implementation of contact. In step 1, grouping the surface pairs in the global contact list is a matter of integer sorting. The BEC environment provides a sort function on a shared array. The parallel section in step 2 is developed via a single program, multiple data (SPMD) model, the same as that used by the BEC technique. Each process accesses "shared" data and does local computation. BEC provides data fetch before computation, thus saving costs associated with fine-grained communication. The fetch is accomplished by registration of the shared data needed, followed by a global exchange of bundled items. Step 3 can be done by concatenation via a prefix sum with copy of purely local data to the shared data structure. BEC will provide a collective operation for this action.

The current contact implementation must support a global view with manual bookkeeping, bundling, and random communication. These are tasks that BEC can do. In Algorithm 1, as in current parallel versions of the contact algorithm, the work of contact *detection* is evenly distributed across processes. It can be seen, by inspection of Figure 14, that contact *enforcement* may suffer from load imbalance.

Algorithm 2 : GCL_partition centric approach

1. In parallel, P_i uses each pair(A,B) of surfaces in GCL_partition[i] to do
    (a) fetch fe(A) and fe(B);
    (b) produce new contact forces;
    (c) put these forces back into FE;
2. In parallel, P_i uses FE_partition[i] to
    (a) propagate the new forces in the finite element mesh;
    (b) produce a new contact sublist CL[i] locally.
3. Assemble the local contact sublists into the global contact list.

Enforcement is naturally load balanced in this algorithm, a point of distinction with Algorithm 1. Figure 15 illustrates communication between shared data structures. Steps 2 and 3 are nearly the same as in the first algorithm, except that contact enforcement takes place from the viewpoint of the global contact list. Registration of all requests for finite element data associated with contact surface pairs is followed by a potentially large exchange of the bundled data. In putting the new forces back onto the FE data structure, the registration of BEC communications is implicit in requests to write to a shared variable, while exchange is explicit.

6.3 Discussion of Potential BEC Approach to Contact Simulation

As we have seen, the global approach that is integral to BEC makes it relatively simple to propose new versions of contact enforcement. The complex nature of the current implementation, as it continues to be developed in the ACME library, makes a literal translation into BEC difficult. The manual bookkeeping and already established communication patterns create a parallel construction that would need to be carefully modified in order to accommodate the BEC library calls.

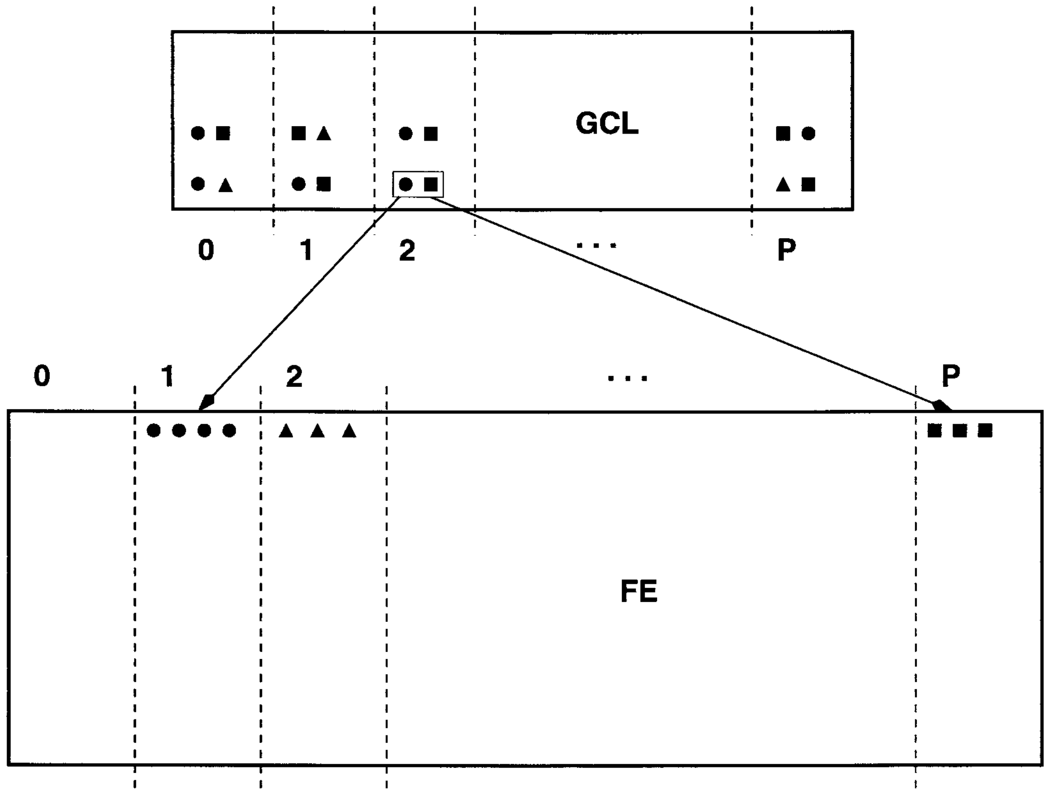

Figure 14. Finite element partition centric algorithm for contact enforcement. This figure illustrates Algorithm 1, in which the decomposition FE_partition is used for contact enforcement. The global contact list (GCL) contains pairs of contact surfaces and geometric information that relates each pair to the physics data in the finite element data (FE). The arrows indicate that information about geometry must flow from the partitioned GCL data structure to the FE_partition. In the example surface contact of the figure, logical processes 1 and P must communicate with each other in order to derive and propagate the contact forces. The GCL_partition is used for parallel contact detection. Note that the same set of P processors is used for both FE and GCL partitions; however, the allocation of data to processors need not correspond between these shared data structures.

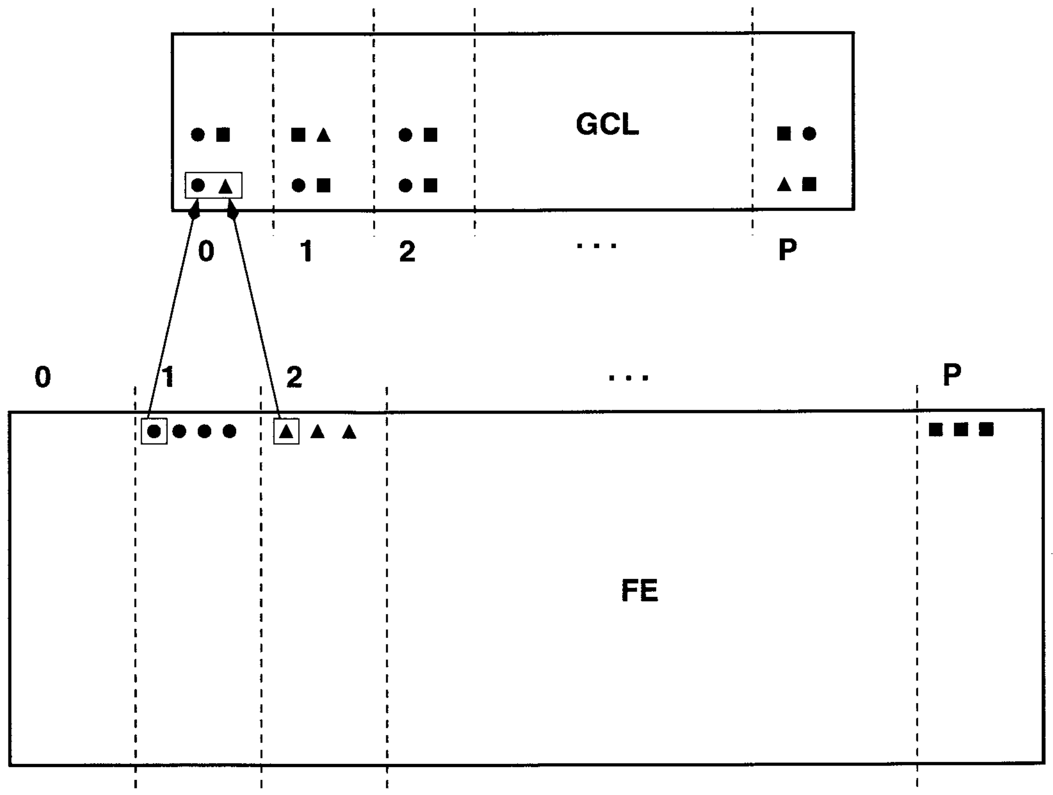

Figure 15. Global contact list centric algorithm for contact enforcement. This figure illustrates Algorithm 2, in which the contact enforcement has been load-balanced by using the global contact list (GCL) processor allocation for this function. The small squares around surfaces in the FE_partition represent additional physical data that must be migrated into the process allocation for the global contact list. It is possible that the data shipment will not be small. The GCL_partition is used in parallel both for contact detection and for contact enforcement. Note that the same set of P processors is used for both FE and GCL partitions; however, the allocation of data to processors need not correspond between these shared data structures.

In the ACME library, there are roughly 75000 source lines of code for contact search; about 10000 lines are related to parallelism in the search. Contact enforcement, by contrast, has approximately 18000 lines, of which about 2000 comprise the manipulation of parallel data structures. Because of the difference in the size of the source code that would require modification for BEC, it makes sense to begin with enforcement. It is also the case that the ACME team is considering alternative strategies for enforcement decomposition. With its capability to abstract communications at a high level, a BEC implementation can hide the parallel bookkeeping tasks, with the potential of more rapid prototyping.

7 Conclusion

We have presented our incremental approach to introduction of new programming models. In particular, we have defined BEC as an abstraction that enables GAS capabilities for parallel programming in SPMD style. It is a portable lightweight approach for the incremental adoption of the GAS programming model. It also provides for some of the unaddressed needs, such as efficient support for high-volume fined-grained and random communications, which are common in parallel graph algorithms, sparse-matrix operations, and large scale simulations.

The BEC abstraction can be implemented as a minor set of language extensions to existing high-level languages such as C and Fortran, plus a thin runtime layer which serves as an interface to the underlying communication infrastructure. Since this runtime layer is so thin, its bulk-transport function can be manually fine-tuned to take advantage of the communication capabilities of a particular platform. The BEC runtime layer can also leverage existing communication utilities that are already optimized. Data from initial experiments with a prototype communication bundling library in the BEC style shows that this approach scales well.

BEC can be used as an enhancement to an existing environment such as MPI. It can serve as an intermediate language [14] to other high level GAS languages such as PRAM C [8] and UPC [30]. Furthermore, it can serve as a bridge between programming models such as virtual shared memory and message-passing.

For future research, there are many directions to extend this work. For example, we are considering the addition of more flexibility to the BEC_exchange operation, which currently behaves as a barrier. Another extension is to allow multiple BEC threads to run on one physical processor.

References

[1] S. W. Attaway, B. A. Hendrickson, S. J. Plimpton, D. R. Gardner, C. T. Vaughan, K. H. Brown, and M. W. Heinstein. A parallel contact detection algorithm for transient solid dynamics simulations using PRONTO3D. *Computational Mechanics*, 22:143–159, 1998.

[2] S. Balay, W. Gropp, L. McInnes, and B. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[3] S. Balay, W. Gropp, L. McInnes, and B. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.22, Argonne National Laboratory, 1998.

[4] S. Balay, W. Gropp, L. McInnes, and B. Smith. PETSc home page. http://www.mcs.anl.gov/petsc, 1998.

[5] Guy E. Blelloch, Michael A. Heroux, and Marco Zagha. Segmented operations for sparse matrix computations on vector multiprocessors. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA., August 1993.

[6] R. B. Brightwell and Z. Wen. Advanced parallel programming models research and development opportunities. Technical Report SAND2004-3485, Sandia National Laboratories, 2004.

[7] J. L. Brown and Z. Wen. Toward an application support layer: Numerical computation in unified parallel c. In *Proceedings of the Workshop on Language-Based Parallel Programming Models*, Poznan, Poland, September 2005.

[8] Jonathan L. Brown and Zhaofang Wen. PRAM C: A new parallel programming environment for fine-grained and coarse-grained parallelism. Technical Report SAND2004-6171, Sandia National Laboratories, 2004.

[9] K. Brown, S. Attaway, S. Plimpton, and B. Hendrickson. Parallel strategies for crash and impact simulations. *Computer Methods in Applied Mechanics and Engineering*, 184:375–390, 2000.

[10] W. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the berkeley UPC compiler. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS 2003)*, 2003.

[11] I. Duff, M. Heroux, and R. Pozo. An overview of the Sparse Basic Linear Algebra Subprograms: The new standard from the BLAS Technical Forum. *ACM Transactions on Mathematical Software*, 28(2):239–267, June 2002.

[12] K. Yelick et. al. Titanium, a high-performance java dialect. *Concurrency: Practice and Experience*, 10:825–836, 1998.

[13] A. Geist et al. PVM home page. http://www.csm.ornl.gov/pvm/pvm_home.html, 2005.

[14] Sue Goudy, Shan Shan Huang, and Zhaofang Wen. Translating a high level PGAS program into the intermediate language BEC. Technical Report SAND2005-xxxx, Sandia National Laboratories, Dec. 2005.

[15] Robert Graybill. High productivity language systems - the path forward (keynote). In *Proceedings of the PGAS Programming Models Conference*, Minneapolis, MN, September 2005.

[16] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 2nd edition, 1999.

[17] M. W. Heinstein, S. W. Attaway, J. W. Swegle, and F. J. Mello. A general-purpose contact detection algorithm for nonlinear structural analysis codes. Technical Report SAND1992-2141, Sandia National Laboratories, Albuquerque, New Mexico, May 1993.

[18] M. W. Heinstein, F. J. Mello, S. W. Attaway, and T. A. Laursen. Contact-impact modeling in explicit transient dynamics. *Computer Methods in Applied Mechanics and Engineering*, 187:621–640, 2000.

[19] Michael A. Heroux. *Epetra Reference Manual*, 2.0 edition, 2002. http://software.sandia.gov/trilinos/packages/epetra.

[20] Michael A. Heroux. Trilinos home page. http://software.sandia.gov/trilinos, 2004.

[21] Jonathan Hill. The Oxford BSP Toolset (url). http://www.bsp-worldwide.org/implmnts/oxtool/.

[22] J. JaJa. *An Introduction to Parallel Algorithms.* Addison Wesley, 1992.

[23] Andrew Johnson. Developing a petascale application using UPC. In *Proceedings of the PGAS Programming Models Conference*, Minneapolis, MN, September 2005.

[24] NPACI. SHMEM tutorial page. http://www.npaci.edu/T3E/shmem.html, 2005.

[25] R. W. Numrich, J. K. Reid, and K. Kim. Writing a multigrid solver using Co-Array Fortran. In *Applied Parallel Computing: Large Scale Scientific and Industrial Problems, 4-th International Workshop*, pages 390–399, 1998.

[26] S. Plimptom, S. Attaway, B. Hendrickson, J. Swegle, C. Vaughan, and D. Gardner. Parallel transient dynamics simulations: Algorithms for contact detection and smoothed particle hydrodynamics. *Journal of Parallel and Distributed Computing*, 50:104–122, 1998.

[27] L.J. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM J. Computing*, 13(2):409–422, 1984.

[28] Ray S. Tuminaro, Michael A. Heroux, Scott. A. Hutchinson, and J. N. Shadid. *Official Aztec User's Guide, Version 2.1.* Sandia National Laboratories, Albuquerque, NM 87185, 1999.

[29] The UPC Consortium. *UPC Language Specification*, 2005.

[30] Berkeley UPC Working Group (url). Berkeley UPC home page. http://upc.lbl.gov, 2005.

[31] Co-Array FORTRAN Working Group (url). Co-Array FORTRAN home page. http://www.co-array.org, 2005.

[32] MPI Forum (url). MPI-2: Extensions to the Message-Passing Interface. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[33] OpenMP Working Group (url). OpenMP home page. http://www.openmp.org, 2005.

[34] SB PRAM Project (url). http://www.ida.liu.se/c̃hrke/fork95.html#Fork95.

[35] The Open Group (url). POSIX home page. http://www.opengroup.org/certification/posix-home.html, 2005.

[36] Leslie G. Valiant. A bridging model for parallel computation. *Comm. ACM*, August 1990.

A BEC Language Specification

A.1 Types

We define the following types:

- `proc_id_t`: The processor ID or rank type
- `partition_object_t`: An *enum* of partition object types. This is built at compile time from the available classes that implement the Partition Object interface.
- `shared_ref_t`: A structure with a processor ID, counter, and offset field. If the shared region was created by a joint allocation, the processor ID is a special value. Otherwise, the processor ID is that of the creating processor. The counter is the value of a physical counter (either a counter for joint or for local allocation calls) on the calling processor.
- `BEC_error_t`: An *enum* of well-known error codes for the BEC object.
- `PlacementStruct`: A structure with a processor ID, offset, and length field.
- `LocalStruct`: A structure with an offset, length, and local pointer.

A.2 Language Extension

A.2.1 The shared Keyword

The `shared` keyword is a type modifier. It is used to declare a shared region. It satisfies

```
shared <type> <identifier> ( [ <integer> ] )*
```

where `<type>` is any valid C type. This declaration is converted by the compiler to a declaration of `shared_ref_t` type. If the declaration is outside functions, it is as if a joint allocation were called and assigned to the identifier, and only one shared region is created. If the declaration is inside of a function, one region will be created per execution of the function.

If `<type>` is a pointer type, that is, a regular C type augmented with a star (*) before the identifier, a shared reference is created. A shared reference behaves just like a pointer, but only supports the operations of indexing (using brackets) and assignment. For example,

```
shared int A[10];
shared int * B;
B = &A[4];
```

The above code results in PROCOF call returning the same value when called with B or A as argument. B and A also agree on and counter value (in their respective shared_ref_t). However, B will have an offset of 4× sizeof(int) greater than that of A.

A.2.2 Pseudo-Functions

The following are wrappers for functions on the BEC interface object. As they respect types, but the BEC interface object is byte-oriented, they are pseudo-functions. The compiler or translator is responsible for scaling when needed. We use <shared_type> to refer to any type that was declared with a shared keyword.

-
  ```
  BEC_error_t BEC_Read(<shared_type> R, unsigned int offset,
                  unsigned int count, void * localPtr)
  ```

-
  ```
  BEC_error_t BEC_Write(<shared_type> R, unsigned int offset,
                  unsigned int count, void * localPtr,
                  bool persistent = false)
  ```

-
  ```
  BEC_error_t BEC_request(<shared_type> R, unsigned int offset,
                  unsigned int length, bool persistent = false)
  ```

-
  ```
  BEC_error_t BEC_exchange(bool clearPersistant = false)
  ```

-
  ```
  shared uint8_t * BEC_local_allocate(size_t totalBytes,
                       partition_object_t identifier, void * hintBuffer)
  ```

-
  ```
  shared uint8_t * BEC_joint_allocate(size_t totalBytes,
                       partition_object_t identifier, void * hintBuffer)
  ```

-
  ```
  bool BEC_is_available(<shared_type> R, unsigned int offset,
          unsigned int length)
  ```

-
  ```
  bool BEC_get_persistence(<shared_type> R, unsigned int offset,
          unsigned int length)
  ```

-
  ```
  BEC_error_t BEC_set_persistence(<shared_type> R, unsigned int offset,
                  unsigned int count, bool bitVal = true)
  ```

-
  ```
  void BEC_clear_persistence(<shared_type> R, unsigned int offset,
          unsigned int count)
  ```

-
  ```
  void BEC_clear_all_persistence_bits(<shared_type> R)
  ```

-
  ```
  void BEC_barrier()
  ```

A.3 BEC Interface Object

The BEC Interface object exposes the following calls. All those involving data offsets and lengths are byte-oriented.

- `BEC_error_t read(shared_ref_t R, size_t offset, size_t length, void * buffer)`
- `BEC_error_t write(shared_ref_t R, size_t offset, size_t length, void * buffer, bool bitVal)`
- `BEC_error_t request(shared_ref_t R, size_t offset, size_t length, bool bitVal)`
- `BEC_error_t exchange(bool clearPersistence = false)`
- `shared_ref_ localAllocate(size_t size, Partition * po)`
- `shared_ref_t jointAllocate(size_t size, Partition * po)`
- `Partition * getPartitionObject(shared_ref_t R)`
- `BEC_error_t setPartitionObject(shared_ref_t R, Partition * po)`
- `RangeQuery * getRangeQueryObject(shared_ref_t R)`
- `BEC_error_t setRangeQueryObject(shared_ref_t R, RangeQuery * rq)`
- `bool isAvailable(shared_ref_t R, size_t offset, size_t length)`
- `bool getPersistence(shared_ref_t R size_t offset, size_t length)`
- `BEC_error_t setPersistence(shared_ref_t R, size_t offset, size_t length, bool bitVal)`
- `void clearPersistence(shared_ref_t R, size_t offset, size_t length)`
- `void clearAllPersistenceBits(shared_ref_t R)`
- `BEC_error_t Initialize()`
- `BEC_error_t Finalize()`
- `BEC_error_t InitializeGlobalMem()`
- `BEC_error_t FinalizeGlobalMem()`
- `Partition * createPartitionObject(partition_object_t T)`
- `proc_id_t PROCOF(shared_ref_t R)`

B Code Examples: Parallel Linked List Ranking

Linked list ranking is a computation that produces a "rank" for each element in the linked list. The "rank" represents an element's "distance" to the end of the list. The last element of the list has rank of 0, while the first element of the list has rank equal to *length of the list* -1.

In a multi-processor setting, a linked list is stored as an array of indexes, each index represents the index of the next element that the current element links to. A fairly efficient parallel list-ranking algorithm utilizes the "pointer jumping" technique. The pointer jumping technique dictates that each element of the list finds the element it points to, `x`, and updates its own link with the link of `x`. With this technique, any element in the list can reach the "top" of the list in $O(logn)$ time, where n is the length of the linked list.

The parallel ranking algorithm is a slight modification to the pointer jumping algorithm. First, we keep ranks of each element in an array `ranks`, and initialize the values to 1, except for the last element (the element with no link to another element), where the rank value is 0. For each round, in addition to modifying the links (pointers) of elements, we also modify their ranks to be the addition of the original rank and the rank of the element they link to. In the $logi\ n$ round, all elements should have no link to chase, and `ranks` should contain the appropriate values.

The following subsection contains the list ranking algorithm in BEC, and the translation to ANSI C by the BEC compiler follows in the next subsection.

B.1 Linked List Ranking in BEC

```
#include <stdlib.h>
#include <math.h>

// shared variables
shared int* ranks; // stores the ranks of each element.
shared int* links; // stores the links of each element.
shared int n;      // list length.

void main ( int argc, char** argv ) {
  int round = 0;
  int i=0;
  int j=0;

  // length of list is user input.
  n = atoi(argv[1]);

  // allocated space for list.  BEC_JointAlloc divides up requested space
// among available processors.
ranks = BEC_JointAlloc(n*sizeof(int));
links = BEC_JointAlloc(n*sizeof(int));

// MAGIC function to initialize links and ranks.
initalizeList();

// number of rounds needed.
round = (int) (log((float) n) / log(2.0));

// main loop.
for (i=0; i<round; i++) {
  // request local values.
  for (j=0; j<n; j++) {
    if ( PROCOF(links[j]) == MY_PID) {
      // for each element this processor handles, request the link and
      // rank value.
      Request(links, j, sizeof(int));
      Request(ranks, j, sizeof(int));
    }
  }
  // fetch data
  BEC_exchange();

  // request (remote) link/rank data.
  // for each element that local element links to, request
  // link and rank data.
  for (j=0; j<n; j++) {
    if ( PROCOF(links[j]) == MY_PID ) {
      // links[j] and ranks[j] always reside on the same processor.
      // if one is local, the other is too.
      Request(links, links[j], sizeof(int));
      Request(ranks, links[j], sizeof(int));
    }
  }
  // fetch data.
  BEC_exchange();

  // update rank
  for ( j=0; j<n; j++) {
    if (PROCOF(links[j]) == MY_PID) {
      ranks[j] += ranks[links[j]];
      links[j] += links[links[j]];
    }
```

```
      }
      // write-through data.
      BEC_exchange();
    }
  }
```

B.2 Compiler Generated C Code

```
#include <stdlib.h>
#include <math.h>

// all shared variables are translated to have shared_ref_t type.
shared_ref_t ranks;
shared_ref_t links;
shared_ref_t n;

void main ( int argc, char** argv ) {
  int round = 0;
  int i=0;
  int j=0;

  // GENERATED DECLARATIONS
  // a variable is generated for each shared variable read.
  int tmp1, tmp2, tmp3, tmp4, tmp5;

  // length of list is user input.
  n = atoi(argv[1]);

  // allocated space for list.  BEC_JointAlloc divides up requested space
  // among available processors.
  ranks = BEC_JointAlloc(n*sizeof(int));
  links = BEC_JointAlloc(n*sizeof(int));

  // MAGIC function to initialize links and ranks.
  initalizeList();

  // number of rounds needed.
  round = (int) (log((float) n) / log(2.0));

  // GENERATED: before shared variable access, we call Initialize() to
  // prepare Shared Memory Manager.
  Initialize();

  // main loop.
  for (i=0; i<round; i++) {
    // request local values.
    for (j=0; j<n; j++) {
      // PROCOF's parameter, links[j], is translated.
      // PROCOF takes a shared_ref_t (links), and an offset (j)
      if ( PROCOF(links, j) == MY_PID) {
```

```
      // Similarly, Request's parameter links[j] and ranks[j]
      // are translated to be a shared_ref_t and an offset.
      Request(links, j, sizeof(int));
      Request(ranks, j, sizeof(int));
    }
  }
  // fetch data
  BEC_exchange();

  // request (remote) link/rank data.
  for (j=0; j<n; j++) {
    if ( PROCOF(links, j) == MY_PID ) {
      // links[j] and ranks[j] always reside on the same processor.
      // if one is local, the other is too.
      Request(links, links[j], sizeof(int));
      Request(ranks, links[j], sizeof(int));
    }
  }
  // fetch data.
  BEC_exchange();

  // update rank
  for ( j=0; j<n; j++) {
    if (PROCOF(links, j) == MY_PID) {
      // GENERATED FOR:
      // ranks[j] += ranks[links[j]];
      // temporary variables hold the values of READ's from shared memory.
      tmp1 = BEC_Read(links, j, sizeof(int));
      tmp2 = BEC_Read(ranks, tmp1, sizeof(int));
      tmp3 = BEC_Read(ranks, j, sizeof(int));
      tmp3 += tmp2;
      BEC_Write(ranks, j, sizeof(int), &tmp3);

      // GENERATED FOR
      // links[j] += links[links[j]];
      tmp4 = BEC_Read(links, j, sizeof(int));
      tmp5 = BEC_Read(links, tmp4, sizeof(int));
      tmp4 += tmp5;
      BEC_Write(links, j, sizeof(int), &tmp4);
    }
  }

  // finalize writing data.
  BEC_exchange();
```

```
    }

    // GENERATED
    Finalize();
  }
```

C Prototype Code Overview

Initial performance numbers were measured on a prototype message aggregation and compression library. The library relied on the calling application to do explicit processing of messages, and did not implement the full Shared Memory Manager interface.

C.1 Library Classes

The prototype library consisted of the following C++ classes:

- BufferPool: A simple pool of buffers, managed with a balanced tree, implementing a very lightweight design pattern.
- Codec: A combination of an Encoder and a Decoder, capturing the notion of a processor-specific mailbox.
- CompressionStatus: An error enumeration with a printer object for capturing error conditions and returning something meaningful to the user.
- Decoder: An object that received an aggregate message and divided it into smaller messages. With a "pull" interface, higher-level code could ask for these smaller messages one-by-one. The Decoder used in benchmarking was based on the ManagedBuffer class.
- Encoder: An object that received small messages and bundled them into aggregate messages. The Encoder offered virtual channels in that it enforced a priority scheme on messages, permitting the placement of small, high priority messages at the start of the aggregate message. The Encoder was also based on the ManagedBuffer class. The Encoder implemented a "push" interface.
- ManagedBuffer: A simple circular buffer that simulated a finite-capacity hardware buffer.
- NetworkLayer: The top-level interface, bundling one Codec per processor along with the MPI exchange code. The NetworkLayer exposed the push and pull functions, with a facility for addressing the remote processors, and "trigger" call similar to the exchange call in BEC.
- SequentialNetwork: A simple implementation of the NetworkLayer implementation that used no MPI code, designed for debugging.

A program using the prototype library would create a NetworkLayer object and interact with it. The NetworkLayer used varying Codecs, Encoders, and Decoders following a strategy design pattern. In addition, a list generator was used that was based on a balanced tree.

C.2 Parallel Ranking Code

The parallel ranking code was implemented with the following major steps:

1. Initialization: Initialize MPI, a NetworkLayer object, and other local variables.
2. List Creation: Generate a "random" list on process 0 and distribute this to the other processors with an MPI_scatter.
3. Make Secondary Structures: Each processor created a Jump and Count array for its portion of the list.
4. Execute a Main Loop: Each processor used pointer jumping and a simulation of the Shared Memory Manager's exchange call to compute the position of each list item in $\log N$ steps.
5. Gather Result and Verify: The result was gathered to processor 0 and checked against a sequential solution.

In the main loop, we simulated Shared Memory Manager behavior by packing data into "data quads" – a struct that tracked the requesting and requested global index in the list, and had additional fields for the values of the remote Jump and Count arrays. As an artifact of how the prototype was implemented, it was natural to use the same structure for all messages. The main loop was as follows:

- The calling code used the requested global index to choose the appropriate processor for a push request of the remote Jump and Count array values.
- Once all requests were enqueued, each processor executed the trigger function of the NetworkLayer to exchange data quads.
- Each processor then received requests for data from the other processors. These were satisfied, and the requesting global index was used to return the data to the requesting processor. The requesting processor ID could also be deduced from the Codec that held the received data quad.
- Once all requests received were satisfied, each processor again called the trigger on the NetworkLayer.
- Finally, each processor unpacked the received data values and updated its local Jump and Count arrays.

DISTRIBUTION:

1 MS 1110
John Aidun, 1435

1 MS 0817
James Ang, 1422

1 MS 0807
Bob Ballance, 4328

1 MS 0817
Bob Benner, 1422

1 MS 0376
Ted Blacker, 1421

1 MS 1110
Ron Brightwell, 1423

1 MS 1110
Jonathan L. Brown, 1423

1 MS 0382
Kevin Brown, 1543

1 MS 0320
William J. Camp, 1400

1 MS 1110
S. Scott Collis, 1414

1 MS 0318
George Davidson, 1412

1 MS 1110
Erik DeBenedictis, 1423

1 MS 0817
Doug Doerfler, 1422

1 MS 0316
Sudip Dosanjh, 1420

1 MS 0817
Brice Fisher, 1422

1 MS 0382
Mike Glass, 1541

1 MS 0817
Sue Goudy, 1422

1 MS 8960
James Handrock, 9151

1 MS 1110
William Hart, 1415

1 MS 0822
Rena Haynes, 1424

1 MS 1110
Bruce Hendrickson, 1414

1 MS 1110
Michael Heroux, 1414

1 MS 0316
Scott Hutchinson, 1437

1 MS 0817
Sue Kelly, 1422

1 MS 0378
Marlin Kipp, 1431

1 MS 1111
Patrick Knupp, 1411

1 MS 0801
Rob Leland, 4300

1 MS 0370
Scott Mitchell, 1411

1 MS 1110
Steve Plimpton, 1412

1 MS 0807
Mahesh Rajan, 4328

1 MS 1110
Rolf Riesen, 1423

1 MS 0378
Allen Robinson, 1431

1 MS 0318
Elebeorba May, 1412

1 MS 0321
Jennifer Nelson, 1430

1 MS 1110
Cynthia Phillips, 1415

1 MS 1110
Neil Pundit, 1423

1 MS 1111
Mark D. Rintoul, 1412

1 MS 1110
Suzanne Rountree, 1415

1 MS 1111
Andrew Salinger, 1416

1 MS 0378
Stewart Silling, 1431

1 MS 0378
James Strickland, 1433

1 MS 0378
Randall Summers, 1431

1 MS 1110
Jim Tomkins, 1420

1 MS 0370
Tim Trucano, 1411

1 MS 0817
John VanDyke, 1423

1 MS 0817
Courtenay Vaughan, 1422

1 MS 1110
Zhaofang Wen, 1423

1 MS 0822
David White, 1424

1 MS 1110
David Womble, 1410

1 MS 0823
John Zepper, 4320

2 MS 9018
Central Technical Files, 8945-1

2 MS 0899
Technical Library, 9616

Second Printing, (March 2006):

1 Dr. Stan Ahalt
205 Dreese Laboratory
Department of Electrical Engineering
The Ohio State University
2015 Neil Avenue
Columbus, OH 43210 USA

1 Dr. David Bailey
Lawrence Berkeley National Laboratory
Mail Stop 50B-2239
Berkeley, CA 94720

1 Dr. Dan Bonachea
777 Soda Hall
Computer Science Division
University of California at Berkeley
Berkeley, CA 94720-1776

1 Dr. Bill Carlson
IDA Center for Computing Sciences
17100 Science Drive
Bowie, MD 20708

1 Dr. Bradford Chamberlain
Cray Inc.
411 First Avenue S, Suite 600
Seattle, WA 98104

1 Dr. Mark Davis
Intel
110 Split Brook Road - SPT 1
Nashua, NH 03062-2711

1 Dr. Jason Duell
777 Soda Hall
Computer Science Division
University of California at Berkeley
Berkeley, CA 94720-1776

1 Dr. Tarek El-Ghazawi
Department of Electrical and Computer Engineering
The George Washington University
801 22nd Street NW 6th floor
Washington DC 20052

1 Dr. Rob Fowler
Department of Computer Science
Rice University
P.O. Box 1892, MS 132
Houston, TX 77251
USA

1 Dr. Al Geist
Oak Ridge National Laboratory
P.O. Box 2008
Bldg 6012
Oak Ridge, TN 37831-6367

1 Dr. Alan George
Department of Electrical and Computer Engineering
University of Florida
PO Box 116200
327 Larsen Hall
Gainesville, FL 32611-6200

1 Dr. Robert Graybill
DARPA
3701 Fairfax Drive
Arlington, VA 22203

1 Dr. Bill Gropp
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439

1 Dr. Joseph JaJa
Institute for Advanced Computer Studies (UMIACS)
A.V. Williams Building
University of Maryland
College Park, MD 20742-3251

1 Dr. Fred Johnson
DOE
SC-31/Germantown Building
1000 Independence Avenue SW
Washington, DC 20585-1290

1 Dr. Laxmikant Kale
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Avenue
Urbana, IL 61801-2302

1 Dr. Ashok Krishnamurthy
Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212

1 Dr. Vipin Kumar
Department of Computer Science and Engineering
University of Minnesota
200 Union Street SE
4-192 EE/CS
Minneapolis, MN 55455

1 Dr. Rusty Lusk
Mathematics and Computer Science Division
Argonne National Laboratory
9700 S Cass Ave
Argonne, IL 60439

1 Dr. John Mellor-Crummey
Department of Computer Science
Rice University
P.O. Box 1892
Houston, TX 77251
USA

1 Dr. Juan Meza
Lawrence Berkeley National Laboratory
Mail Stop 50B-2239
Berkeley, CA 94720

1 Dr. Phil Merkey Department of Computer Science
Michigan Tech University
1400 Townsend Dr.
Houghton, MI 49931

1 Dr. Robert W Numrich
Minnesota Supercomputing Institute
University of Minnesota
599 Walter Library
117 Pleasant St. SE
Minneapolis, MN 55455

1 Dr. Steve Reinhardt
SGI
2750 Blue Water Road
Eagan, MN 55121

1 Dr. Vivek Sarkar
IBM Research
T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598

1 Dr. P. Sadayappan
Department of Computer Science and Engineering
595 Dreese Lab
2015 Neil Avenue
Ohio State University, Columbus, Ohio 43210 USA

1 Dr. Steve Seidel
Department of Computer Science
Michigan Tech University
1400 Townsend Dr.
Houghton, MI 49931

1 Dr. Lauren Smith
NSA
1806 Stonegate Ave.
Corfton, MD 21114

1 Dr. Marc Snir
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Avenue
Urbana, IL 61801-2302

1 Dr. Guy Steele
Sun Microsystems
1 Network Drive
Burlington, MA 01803

1 Dr. Thomas Sterling
Center for Computation and Technology
Department of Computer Science

202 Johnston Hall
Louisiana State University
Baton Rouge, LA 70803

1 Dr. Uzi Vishkin
Institute for Advanced Computer Studies (UMIACS)
A.V. Williams Building
University of Maryland
College Park, MD 20742-3251

1 Dr. James (Trey) White III
Oak Ridge National Laboratory
1 Bethel Valley Road
PO Box 2008 MS-6008
Oak Ridge, TN 37831-6008

1 Dr. Brian Wibecan
UPC Development Team
Hewlett-Packard Company
110 Split Brook Road
ZKO1-3/D40
Nashua, NH 03062-2698

1 Dr. Kathy Yelick
777 Soda Hall
Computer Science Division
University of California at Berkeley
Berkeley, CA 94720-1776

1 Dr. Thomas Zacharia
Oak Ridge National Laboratory
1 Bethel Valley Road
Box 2008
Oak Ridge, TN 37831

1 Dr. Hans Zima
Principal Scientist
JPL
Caltech
4800 Oak Grove Drive MS 171-373
Pasadena, CA 91109