SAND2008-2751C

The BEC Programming Model

Mike Heroux
Sandia National Labs
Albuquerque, NM 87185
Email: maherou@sandia.gov

Zhaofang Wen
Sandia National Labs
Albuquerque, NM 87185
Email: zwen@sandia.gov

Junfeng Wu
Syracuse University
Syracuse, NY 13244-1150
Email: juwu@syr.edu

Abstract—**Bundle-Exchange-Compute (BEC) is a programming model supporting PGAS on distributed-memory machines. The BEC programming abstraction incorporates the spirits of two famous computation models: PRAM [1], [2] (with random fine-grained accesses to shared memory) for its expressiveness of parallelism, and BSP [3], [4] for its efficiency on distributed-memory machines.**

BEC's global view of shared data structures enables ease of algorithm design and programming. For good performance, the BEC model guides the programs to be written in an efficient BSP-like style. Fine-grained shared data accesses are automatically and dynamically bundled together for coarse-grained message-passing. The BEC model allows overlap communication and computation, and provides ample optimization opportunities for communication scheduling inside the BEC runtime library.

BEC code can mix with MPI code. Integrating BEC with MPI is an easy way to add PGAS programming abstraction to MPI. Integrating BEC with MPI or other PGAS languages can add built-in capabilities to these models for efficient support of unstructured applications.

BEC is easy to use with a simple memory model (no race condition). General users are free from explicit management of data distribution, locality, communication, synchronization. Advanced users can specify arbitrary data distribution. Initial unstructured applications using BEC show that simple BEC programs can match very complex and highly optimized MPI codes in performance.

I. Introduction

Bundle-Exchange-Compute (BEC [5], [6]) [1] is a programming model supporting Partitioned Global Address Space (PGAS) [7] on distributed-memory machines. The BEC programming environment extends ANSI C with shared variables, and includes a light-weighted runtime library (called BEC Lib) on top of the message-passing layer (e.g. MPI).

BEC is easy to use with a simple memory model (no race condition) and a somewhat higher level of programming abstraction than current PGAS languages. General users are free from explicit management of data distribution, locality, communication, synchronization. Advanced users can specify arbitrary data distribution. Details of the BEC model as well as its role in relation to MPI and other PGAS languages are presented later in Section VI. Initial applications using BEC have shown that very simple BEC code can match complex and highly tuned MPI code in performance. (These applications are difficult to implement using other PGAS languages to achieve compatible performance with MPI.)

[1]developed at Sandia National Labs in collaboration with Syracuse University

BEC is developed based on (1) in-depth analysis of HPC applications at Sandia National Labs and their programming needs, and (2) careful study and comparison of past and present practical parallel programming models (e.g. Open MP, MPI, PGAS etc. [8], [9], [7]) as well as theoretical parallel computation models (e.g. PRAM [2], [1], BSP [3], [4]).

II. General Overview of Parallel Programming Languages and Libraries

A parallel programming model provides an abstraction for programmers to express the parallelism in their applications while simultaneously exploiting the capabilities of the underlying hardware architecture. A programming model is typically implemented in a programming language, or a runtime library, or both; and the implementation is also referred to as a programming environment.

For decades researchers and language developers have been exploring and proposing parallel library and language (PLL) extensions to support large-scale parallel computing. In the entire time, MPI has been the only project that can be called a broad success. PVM [10] and SHMEM [11] have made an impact on a subset of platforms and applications. Shared memory parallel models such as POSIX [12] Threads and OpenMP [8] 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 slower than its sequential counterpart. Ironically, the success of Message Passing Interface (MPI [13]) has made the adoption of true language extensions and other novel library approaches extremely difficult across existing parallel applications bases and with 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 address this attitude. There are still many opportunities to improve upon MPI, both in usability and performance.

Despite MPI's success, there are still classes of applications for which this model is not best suited, for example, parallel graph algorithms [14]. Furthermore, some studies suggest that programmer productivity can be higher with other programming models such as PGAS [15], [16], [17]. The DARPA

HPCS program is in its Phase III (2006-2010) with a focus on programming models [18]. PGAS is regarded as a key step towards the HPCS goals. Major parallel machine vendors have dedicated teams developing their own future models (e.g. IBM's X10 [19] and Cray's Chapel [20]), all to offer PGAS as a subset. PGAS models can be realized in libraries and language extensions. Examples of PGAS libraries are MPI-2 [21], and Cray's SHMEM [11]. Existing PGAS language extensions include Unified Parallel C (UPC) [15], Co-Array Fortran (CAF) [22], and the Java-based Titanium [17].

There is also the question whether the message-passing model (alone) will be suitable for new architectures in coming decades. With the overwhelming complexities of the next-generation parallel systems based on multi-core chips and heterogeneous architectures, higher level programming abstraction with better ease of use will be necessary.

III. Technical Criteria

A practical programming model must balance the often conflicting technical factors including **application performance** (arguably the most important), **ease of use**, performance **scalability** to increasing number of processors, and **portability** to a wide range of architectures and platforms. Some of these criteria are self-explanatory; while ease of use is discussed in more details below.

The phrase "ease of use" have had many interpretations, which can make it confusing and hard to compare and judge programming models. So it is important to clarify what this phrase means. Specifically, ease of use should be reflected in the process of the application program development, as outlined below.

1) *Design parallel algorithms to exploit the inherent parallelism* in the application problem formulation. Given a problem, there can be many dramatically different and "fast" parallel algorithms, based on various high-level parallel computation models (e.g. SIMD shared memory fine-grained parallel [1], [2], distributed memory message-passing coarse-grained parallel [21], [10]).
2) *Map (implement) the algorithm(s) onto a specific level of programming abstraction*. There are several fundamental issues here.
   a) *Expressiveness of Parallelism:* The abstraction level of the programming interface should allow easy and natural expression of parallelism in the parallel algorithms. Such parallelism may be coarse-grained or fine-grained, and involve regular (remote) data access patterns or random data accesses. For example, MPI programs often need a support layer, either domain-specific libraries or user-written functions, in order to implement fine-grained parallelism with irregular data accesses.
   b) *More Algorithm Choices:* Some of the algorithms may be readily implementable, or implementable with minor adjustments; some others may require a lot of extra support (e.g. another software layer), or may simply be unsuitable. Therefore, "ease of use" should also include support for more algorithm choices.
   c) *Guidance to Good Program Style:* For ease of use, a programming model should implicitly and gently guide the mapping of parallel algorithms into parallel programs with good-style (e.g. BSP style [3], [4]) that can run efficiently on the target architecture. This is the so-called **easy to do good programming**. In the contrary, **easy to do bad programming** means that a programming model provides the programming convenience (a "trap") that may result in severe performance penalty.
   For example, some PGAS language allows random accesses to elements of shared arrays that actually are always distributed in fixed regular patterns across the physical processors. This programming feature makes it convenient to express parallel algorithms with irregular shared data accesses. But such a convenience can potentially lead to programs with very poor performance; and the unrestricted fine-grained random accesses to shared array elements in this programming abstraction has the risk of producing programs with such arbitrary styles that are very hard for optimizations in the compiler and runtime library.
3) *Performance Tuning:* It may be easier to write a quick version of a parallel program using a high-level programming abstraction than a low level one. But the quick high-level version may not always provide the desirable performance. One question for "ease of use" here is whether or not the high-level programming abstraction can be retained in performance tuning. Or does the high-level abstraction have to be replaced with low-level abstraction in order to get good performance?
   For example, in some PGAS language, the high-level PGAS abstraction (shared arrays and related constructs) often has to be abandoned and replaced with low-level one-sided message passing utilities (e.g. $memget()$ and $memput()$) in order to achieve comparable performance to MPI. Performance tuning often leads to a final program of message-passing style. Such cases may indicate insufficiency in the language's PGAS abstraction, and undermines its intended goal for ease of use.

IV. Focus

Our research focuses on the fundamental aspects of a programming model rather than the software engineering aspects of an integrated development environment. The fundamental aspects include easy expression of parallelism (fine-grained and coarse-grained, structured and unstructured etc.) and efficient mapping/implementation of the parallelism onto the machine platforms. The software engineering aspects include, for example, support for Object-Oriented design and coding, strong typing etc.

V. Technical Motivations of the BEC Project

BEC was motivated by two fundamental and un-addressed (or under-addressed) needs.

- *Built-in efficient support of unstructured applications:* These applications inherently require high-volume, random, fine-grained communication (or remote data accesses). For example, unstructured application include parallel graph algorithms, sparse-matrix algorithms, and material physics simulations. Unstructured applications represent a large percentage of parallel applications at Sandia National Labs (at least 50% by some estimate). Furthermore, the theme of the cutting edge research in scientific computing is about developing stable fast algorithms and adaptive methods, such as multi-scale wavelets, kernels, finite volumes, and adaptive finite elements, all of which ultimately involve large, parallel solution of unstructured sparse linear systems.
- *Smooth migration of legacy MPI applications and their programmers to the PGAS model:* For example, at Sandia National Labs, legacy MPI applications represent an on-going multi-billion dollar investment. Serving as the backbone of Sandia's critical missions, these applications can not be abandoned for complete redevelopment; meanwhile, their developers represent most of the expert parallel programmers in the field of HPC today. This means that migration of heavily-invested legacy applications and their expert programmers, along with the costs of development and adoption of new models, pose real challenges to any new programming model effort.

VI. BEC Overview

BEC is a parallel programming model and environment. BEC can be used alone. But equally interesting is that BEC code can mix with MPI code in the same program or even the same function. Integrating BEC with MPI as an enhancement can add PGAS capability to MPI as well as built-in capabilities for management (bundling/unbundling) and efficient scheduling of communication; this should help the smooth migration from MPI to higher level models. BEC can also be integrated with existing PGAS language implementations to provide built-in efficient support for unstructured applications. (In fact, there has been discussion to do so with UPC [8].) Furthermore, BEC or the capabilities of BEC) can potentially serve as building blocks in next-generation parallel programming languages. Finally, BEC can also function as an intermediate language [23] to high-level PGAS programming language constructs [24].

The BEC programming model incorporates the spirits of two famous parallel computation models: the Parallel Random Access Model (PRAM, [2], [1]) with shared memory for its expressiveness of random fine-grained parallelism / algorithms, and the Bulk Synchronous Parallel (BSP, [3], [4]) model for its efficiency when implemented on distributed memory machines. Specifically, BEC's global view of shared data structures enables ease of algorithm design and programming. For good performance, the BEC model guides the programs to be written in BSP-like style, in which fine-grained shared data accesses are automatically and dynamically bundled together (by the runtime library to schedule for) coarse-grained message-passing.

The BEC model also makes it easy to overlap communication and computation. The BEC model also provides ample optimization opportunities for message packing/unpacking and communication scheduling inside the runtime library; otherwise such optimizations are often written by individual MPI programmers in an ad hoc fashion.

BEC is easy to use with a simple memory model (no race condition). General users are free from explicit management of data distribution, locality, communication, synchronization. Advanced users can specify arbitrary data distribution. Initial experiences show that **BEC satisfies all the technical criteria discussed earlier in Section III.**

The strength of BEC is most apparent for unstructured applications that inherently require high-volume random fine-grained communication, such as parallel graph algorithms, sparse-matrix operations, and large scale simulations.

As a parallel programming environment, BEC extends ANSI C with shared variables to support parallel programming, and includes a light-weighted runtime library (called BEC Lib) on top of the message-passing layer (e.g. MPI). The BEC runtime library API has a small set of functions to let users program in the Bundle-Exchange-Compute style. This runtime library can be used alone or together with the BEC programming language extension. The BEC language extension is convenient because it allows shared arrays to be used in regular array syntax, which also allows the BEC library functions to be used more conveniently. Advanced users who prefer to use BEC Lib directly with C or C++ may be able to write even more efficient codes. In either case, the Application Programming Interface (API) of the BEC runtime library is fairly simple. A user only needs to know a few functions. A BEC program typically include one or more Bundle-Exchange-Compute phases as follows. [2]

1) **Bundle:** accesses to shared data (variables) must be explicitly requested before they can be used in local computation. These requests are automatically and dynamically aggregated into a bundle object by the BEC runtime library. Write requests to shared variables can either be made explicitly by calling BEC runtime library functions, or implicitly in the BEC language extension in which case assignment statements to shared variables are translated into BEC runtime library calls for write requests. The BEC runtime library offers functions for both of these purposes. It also provides functions to create a persistent bundle object for multiple bundle-exchange-compute phases.

[2]With advanced compiler support, it is possible to generate the read requests automatically. A user only needs to insert some "BEC_exchange()" calls in the "right place" in between the computation code.

2) **Exchange:** a call to function *BEC_exchange()* tells the BEC runtime library to make sure that the bundles are exchanged among the physical processors to fulfill the read and write requests of shared data. This is a collective operation. Depending on implementation of the BEC runtime library, the actual exchange (transfer) of data may occur before *BEC_exchange()* is called. However, it is only after this call that it is safe to assume that the requested data are available for computation.
3) **Compute:** After the exchange, shared data can be used as if they were local. Actual requests to write shared variables are made (explicitly or implicitly) in this step. These requests will be bundled up with the read requests in the next phase of Bundle-Exchange-Compute.

For example, consider the following code segment.

```
shared int A[10000], B[10000], C[10000];

BEC_request(A[3]);
BEC_request(B[8]);
  /* explicitly request A[3], B[8]  */

BEC_exchange();

C[10] = A[3]+B[8];
  /* Use A[3], B[8] in computation and*/
  /*request to write C[10] implicitly.*/
```

NOTE:

- The BEC exchange step is a collective call. It resolves all the pending (write and read) requests to shared memory locations.
- If multiple write requests are made to the same shared location, predetermined rules of the BEC runtime library decide which one to succeed.
- When a processor reads a shared location, if there is a write request by the same processor after the preceding exchange call, the read operation will get the value of that write request; otherwise, it will get the value of the shared location at the end of the preceding exchange call.

VII. Applications and Performance

We have implemented several initial applications using BEC. Three of these applications are presented here for demonstration. These three applications are

- a simple random access benchmark that is somewhat similar to the HPC Challenge Global Random Access benchmark,
- a graph coloring algorithm based on the Largest Degree First (LDF) heuristic, and
- a sparse linear system solver using the Conjugate Gradient (CG) method.

The first application is a test, and it is chosen to show the impact of bundling to application performance vs. no bundling. Implemented in both BEC and MPI for comparison, the other two applications show that BEC is significantly easier to use than MPI while achieving comparable application performance; and this is because BEC has built-in bundling capabilities while the MPI programs need to include additional code for bundling in ad hoc fashions.

A. Summary of Inital Experiences Using BEC

Experiences from implementing these three can be summarized as the following:

1) BEC can support unstructured applications very efficiently (as intended).
2) With a few lines of BEC code automatically invoking its built-in bundling, it can achieve very good application performance that would require very complex MPI codes to match (to our surprise).
3) BEC's PGAS allows easy algorithm design and expression of parallelism beyond MPI; and its built-in dynamic bundling concept provides capabilities for efficient implementation (a capability not yet available in existing PGAS languages). This combination can overcome programming difficulties that prevent many applications from becoming MPI parallel applications, because it frees application programmers from system level details unrelated to their own domain expertise.

Finally,the built-in efficient support for random fine-grained shared data accesses raises the level of PGAS programming abstraction, and advances the state of the art. BEC can help smooth migration from MPI to PGAS programming models.

B. Parallel Machine Used

All performance charts are made according to the test results collected from the NERSC supercomputer Franklin (franklin.nersc.gov). Franklin is a Cray XT4 system. More detail about Franklin can be found at http://www.nersc.gov/nusers/resources/franklin/.

C. Global Random Access

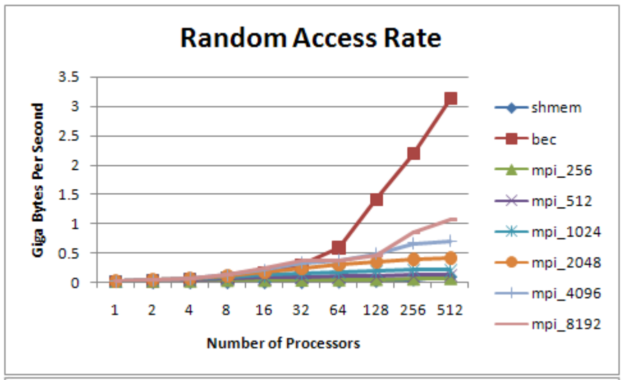
The test here is to show the importance of message bundling to application performance when compared to no-bundling. This test is similar to the HPCS GUPS benchmark [25]; but there are some differences, which will be discussed later in this section.

Specifically, the problem involves a shared array *X[N]*, distributed equally over these P processors, with each processor holding a contiguous section of N/P items. For the test, each processor to update the shared array for M rounds. In each round, every processor randomly selects N/P items of the shared array to update.

Algorithm:

```
for (round = 0; round < M; round ++) {
  for (i = 0; i < N / P; i ++) {
    index = random() % N;
    value = random();
   X[index] = value;
  }
}
```

The first chart in Figure 1 shows the rates of random accesses by various models, in terms of Giga Bytes Per Second, on various numbers of processors. The second chart



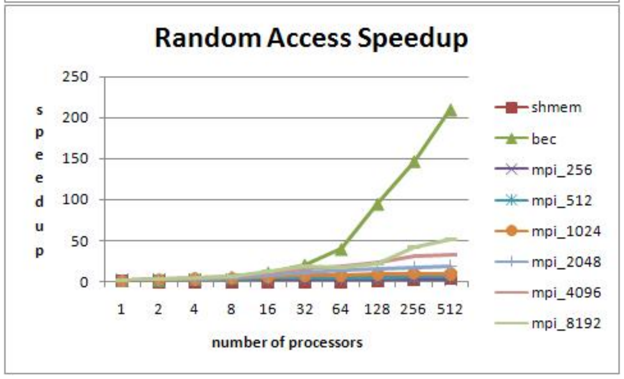


Fig. 1. Performance of Random Access

in Figure 1 shows the scaling of these rates relative to the number of processors. In these charts, for example, the curve labeled "mpi_256" represents the performance of the MPI implementation that allows bundling up to 256 items. This number is also called lookahead size in the HPCC Global-Random-Access benchmark.

As shown in Figure 1, the BEC program scales better than the other two programs. The scaling of the MPI program depends on the lookahead size. Increasing the lookahead size improves the granularity of the communication of the MPI program, thus improves its performance and scaling. This further demonstrates the importance of bundling. With MPI, such bundling requires substantial extra coding efforts in the users' part; while with BEC, bundling is implicit and automatic with no extra coding effort.

The HPCC Random Access benchmarks are measured in term of Giga Updates Per Second (GUPS). These benchmarks are used for the ranking of supercomputers for the their abilities to support applications that require random or irregular data accesses.

Although the GUPS benchmark has been used mainly to test the hardware platform, a real application is developed on a machine platform comprising hardware and system software (including programming environment); so it would be meaningful (even more so for both practical reason and for the reaching the DARPA goal) to test the GUPS on the programming environment on which the real application is developed. In this context, we choose to relax the restriction of the size of the look-ahead for two reasons: (i) in real application development, the message-queues used by programmers are much less restrictive; and (ii) more importantly, BEC has built-in message-bundling capabilities that are automated and basically requires no effort on the programmers' part to handle random accesses, regardless of the sizes of the message-bundles.

D. Graph Coloring Algorithm Based on the Largest Degree First Heuristic

Graph algorithms have many applications in high performance parallel computing, especially in optimizations , simulations, and even the parallelization of traditional numerical methods such as Gauss-Seidel iterative method. For example, graph coloring is used in the parallel Gauss-Seidel method.

For our applications here, we choose the vertex coloring, which is to assign colors to vertices in a graph such that no two adjacent vertices share the same color. The specific algorithm is based on a heuristic called Largest Degree First (LDF) as described below.

Input: G — the graph with n vertices *v_1, v_2, . . . , v_n*
Output: c(v_1), c(v_2), . . . c(v_n) — the assigned colors
Algorithm:

```
Randomly assign weight to every vertex on
    this processor;
while(there are uncolored vertices in G) {
  for each uncolored vertex on this
      processor {
    mark the vertex as a candidate;

    for each uncolored neighbor of this
        vertex {
      if (the degree of neighbor > the
          degree of this vertex) {
        unmark the vertex;
        break;
      }
      if (the degree of neighbor == the
          degree of this vertex && the
          weight of neighbor > the weight
          of this vertex) {
        unmark the vertex;
        break;
      }
    }

    if (the vertex is still marked) {
      iterate through its uncolored
      neighbors to find out the smallest
      possible color;
      assign this smallest possible color
      to the vertex;
    }
  }

  for each picked vertex v_i {
    c(v_i) = smallest possible color;
  }
}
```

Note that accessing the neighbors (traversing the edges) for each uncolored vertex typically cause irregular (and potentially remote) data accesses. Without bundling, these data accesses will cause high-volume random fine-grained communication. As shown in the random access test discussed earlier, this kind of communication can significantly hurt application performance on distributed memory machine platforms.

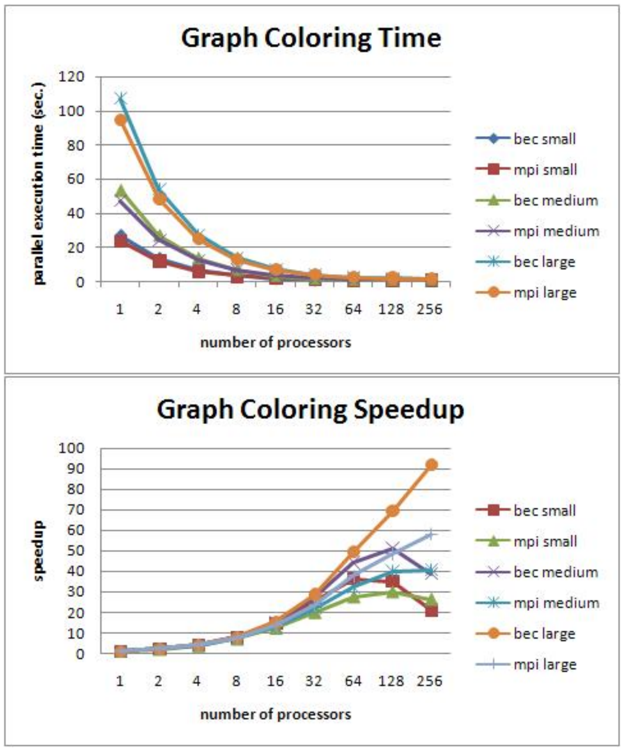


Fig. 2. Performance of Graph Coloring

The first chart in Figure 2 shows the parallel execution time of both BEC and MPI implementations of the graph coloring algorithm. The second chart in Figure 2 shows the performance scaling of the BEC and MPI implementations relative to the number of parallel processors. The three different curves for BEC as well as those for MPI represent the performance of the programs on data sets of three different sizes (small, medium, and large). The small data set contains 5 million vertices and about 80 million edges, the median one contains 10 million vertices and about 16 million edges, while the large one contains 20 million vertices and about 32 million edges.

As shown in Figure 2, the performance of the BEC and the MPI implementations are comparable. The BEC implementation started out a little bit slower on low processor count. This is because the runtime support of the shared variables incurs some overheads in BEC, while the MPI implementation is distributed and therefore without these overheads (the draw back is the need for much more codes). As the processor count increases, the BEC implementation catches up relative to the MPI implementation. This is because the BEC runtime library's built-in data structures (such as hash tables) are specially optimized for larger processor count, while the MPI implementation uses general purpose hash tables from the C++ Standard Template Library (STL), which is not optimized in this way. In Figure 2, BEC seems to scale better as the number of processors increases, for two possible reasons: (1) The BEC implementation started out slower. (2) as the processor count increases, the highly-optimized and specialized hash tables inside BEC Lib provides better performance than the general purpose hash tables of the STL used by the MPI program; and such performance advantage well compensates for BEC Lib's overheads visible at low processor count. The BEC and MPI implementations have similar performance scaling trend for the same input data set.

task	BEC lines	MPI lines
communication (bundling included)	10	69
computation	58	61
whole program	131	201

TABLE I
CODE SIZES OF THE GRAPH COLORING PROGRAMS

Table I shows the comparison of the BEC and MPI implementations in terms of code sizes in various parts of the graph coloring kernel. The ratio of code size for bundling and exchange to the code size for computation is about 0.17 for BEC, and more than 1 for MPI. The reasons for the simpler BEC code are as follows.

- The BEC user only needs to write a few lines of code to request the shared data regardless of their physical locations. Its built-in bundling and associated hashing are invoked automatically and implicitly. The requested data becomes available after a single call to the *BEC_exchange()* function.
- The MPI user, in contrast, has to write code for creating message queues, ad hoc data packing and unpacking, and hashing data for their repeated use in computation.

It is worth pointing out that if the MPI program does not use the C++ STL for its hashing capabilities, the MPI user will have to write additional code to implement the hash tables, which not only is non-trivial to do for most users, but also can significantly increase the already larger code size of the MPI implementation.

E. Sparse Linear System Solver Using the Conjugate Gradient Method

This application is a linear system solver using the Conjugate Gradient (CG) method on an arbitrary number of processors. The linear system solved in this program is from the diffusion problem on 3D chimney domain by a 27 point implicit finite difference scheme with unstructured data formats and communication patterns. The sparse-matrix vector multiplication in this CG method is described below.

Input:

x — the shared array for the input vector

A — the sparse matrix

Output:

y — the shared array for the output vector

Algorithm:

```
for each y[i] on this processor {
  y[i] = 0;
  for each nonzero A[i][j] {
    y[i] += A[i][j] * x[j];
  }
}
```

An MPI program is used for the comparison to the BEC CG program. This MPI program was written by Mike Heroux, and used as a micro application for research in many areas for several years at Sandia. Both the BEC and the MPI codes are highly tuned, and both solve the problem in two parts:

1) Bundle preparations (setting up message queues, packing and unpacking data, localizing matrix, etc), and
2) CG iterations (until convergence).

The performance comparison is shown in Figure 3. We use notation "BEC_64_64_512" to represent the performance of BEC on a 3D chimney domain (64h, 64h, 512h), where "h" is lattice size in space (similarly for MPI). For this case, The sparse matrix involved in the algorithm is (64*64*512) rows by (64*64*512) columns. Similar notation is also applied to the MPI program.

As shown in Figure 3, the BEC and MPI have compatible performance. The better speedup of BEC than MPI is partly due to the fact that BEC started out slower than MPI. Theoretically, we do not expect the BEC version to run faster than the MPI version; but we see the BEC version to be slightly faster in some cases, for the following possible reasons:

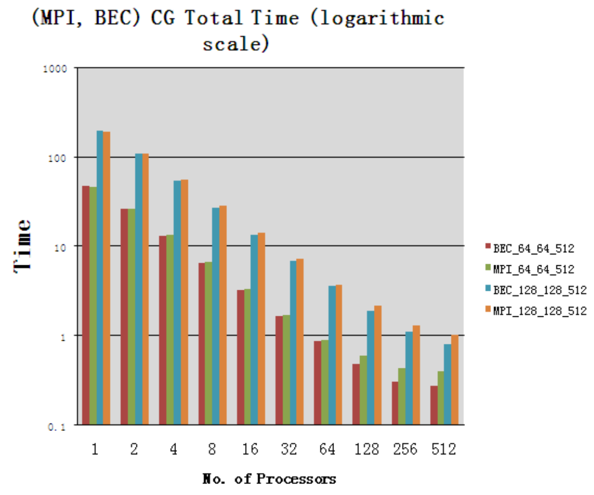
- The MPI code uses a hashing capability from the C++ STL, which may be suboptimal for this specific need. In contrast, the built-in hash table in BEC lib is well optimized. **However, if the MPI version is to write a customized optimal hash table, it can further add to its code complexity.**
- In the CG iteration (computation) part, the MPI code uses blocking MPI "send/receive"; while the BEC runtime library internally uses non-blocking "send/receive" followed by "wait".

These performance comparison results are similar to those of the graph coloring application, except that the CG programs are affected less by the communication overheads here. It is possibly because of the better communication pattern of CG application than the graph coloring application.

The code-size comparison is shown in Table II. Empty lines, comment lines, debugging code lines and # lines are not counted. The reason for why the BEC program is simpler than the MPI program in this CG application is similar to that in the graph coloring application.

task	BEC lines	MPI lines
bundle preparation	6	240
CG iterations	60	87
communication	11	277
whole program	233	733

TABLE II
CODE SIZES OF THE CG PROGRAMS



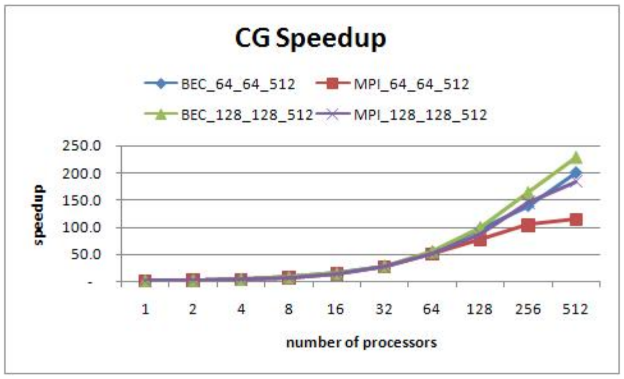


Fig. 3. Performance of Conjugate Gradient

VIII. CONCLUSION

We have presented the BEC programming model. This model satisfies all the technical criteria described in Section III, especially good application performance and ease of good programming.

We have also presented results from some BEC applications. Initial experiences from these BEC applications are summarized below.

1) BEC can support unstructured applications very efficiently (as intended).
2) With a few lines of BEC code automatically invoking its built-in bundling, it can achieve very good application performance that would require very complex MPI codes to match (to our surprise).
3) BEC's virtual shared memory allows easy algorithm design and expression of parallelism beyond MPI; and its built-in dynamic bundling concept provides capabilities for efficient implementation. This combination can overcome programming difficulties that prevent many applications from becoming MPI parallel applications, because it frees application programmers from system level details unrelated to their domain expertise.

Although BEC can be used alone, BEC can be integrated with MPI as enhancement to add built-in PGAS programmability and built-in communication bundling and efficient scheduling capabilities to support unstructured applications. BEC can also be integrated with other PGAS languages such as UPC to provide built-in efficient support for unstructured applications. Such added capability integrations bring MPI and the PGAS model closer, and can facilitate smooth transitions of MPI applications and programmers to the PGAS model.

Finally, the high-level programming abstraction of BEC free the users from the usual parallel programming difficulties (as endured by MPI programmers) in explicit management of data distribution, data locality, communication scheduling, and synchronization. With the overwhelming complexities of the next-generation parallel machines based on clusters of nodes with multi-core chips (with multiple levels of memory) and heterogeneous architectures, a high level programming abstraction that can free users from the usual parallel programming difficulties will become necessary in order for most programmers to make good use of the machines.

Acknowledgement

This research was sponsored 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.

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.

Jonathan Brown, Shan Shan Huang, Sue Goudy and Yuesheng Xu made many contributions to the research efforts that led to the BEC programming environment. Ron Brightwell helped a lot in seeking funding support for this research. Weicai Ye did a lot of work in data collection, test case preparation, and information gathering. We would like to express our appreciation to Bill Camp for his constructive criticism, encouragement, and support for our programming model research. Many thanks to Danny Rintoul for supporting the BEC research through CSRI DT. 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 some related references. Many thanks also to Rolf Riesen, Kevin Petretti, Mike Glass, Kevin Brown, Courtenay Vaughan, Bruce Hendrickson, Steve Plimpton, Doug Doerfler, Brice Fisher, David Burnholdt, Maya Gokhale, and Ron Oldfield.

The BEC model was inspired by the PRAM model [10] and algorithms research of 1980s- 1990s, the BSP model [13], and also more recent work in Global Address Space. We would like to acknowledge the leaders for their contributions. These notably include Uzi Vishkin, Leslie Valiant, Bill Carlson, Kathy Yelick, and Bob Numrich. In particular, we gratefully acknowledge Uzi Vishkin and Bill Carlson for their encouragement. Thanks also to the UPC Consortium and, in particular, Lauren Smith, Tarek El-Ghazawi, Phil Merkey, Steve Seidel, and Dan Bonachea.

References

[1] R. M. Karp and V. Ramachandran, *Parallel algorithms for shared-memory machines*. MIT Press, Cambridge, MA, 1991, pp. 869 – 941.
[2] L. Stockmeyer and U. Vishkin, "Simulation of parallel random access machines by circuits," *SIAM J. Computing*, vol. 13, no. 2, pp. 409–422, 1984.
[3] L. G. Valiant, "A bridging model for parallel computation," *Comm. ACM*, August 1990.
[4] J. Hill, "The Oxford BSP Toolset (url)," www.bsp-worldwide.org/implmnts/oxtool/.
[5] BEC home page, http://www.cs.sandia.gov/BEC/.
[6] Z. Wen, J. Wu, and Y. Xu, "BEC specification and programming reference," Sandia National Laboratories, Albuquerque, NM, Tech. Rep. SAND2007-7617, 2007.
[7] R. B. Brightwell and Z. Wen, "Advanced parallel programming models research and development opportunities," Sandia National Laboratories, Tech. Rep. SAND2004-3485, 2004.
[8] O. A. R. B. (url), "OpenMP fortran application interface version 1.1," www.openmp.org.
[9] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 2nd ed. MIT Press, 1999.
[10] A. Geist *et al.*, "PVM home page," 2005, www.csm.ornl.gov/pvm/pvm_home.html.
[11] NPACI, "SHMEM tutorial page," 2005, www.npaci.edu/T3E/shmem.html.
[12] T. O. G. (url), "POSIX home page," 2005, www.opengroup.org/certification/posix-home.html.
[13] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI-The Complete Reference, Volume 1, The MPI core*. The MIT Press, 1998.
[14] B. Hendrickson, "Combinatorial scientific computing: The role of discrete algorithms in computational science and engineering," 2003, plenary talk at 2nd SIAM Conf. Computational Science & Engineering CSE'03.
[15] U. Consortium, "UPC language specification (v 1.2)," http://www.gwu.edu/ upc/documentation.html.
[16] C.-A. F. W. G. (url), "Co-Array FORTRAN home page," 2005, www.co-array.org.
[17] K. Y. et. al, "Titanium, a high-performance Java dialect," *Concurrency: Practice and Experience*, vol. 10, pp. 825–836, 1998.
[18] R. Graybill, "High productivity language systems - the path forward (keynote)," in *Proceedings of the PGAS Programming Models Conference*, Minneapolis, MN, September 2005.
[19] IBM, "The X10 Programming Language," http://domino.research.ibm.com/comm/research_projects.nsf/pages/x10.index.html.
[20] C. (url), "Chapel — The Cascade High-Productivity Language," http://chapel.cs.washington.edu/.
[21] M. (url), "MPI-2: Extensions to the Message-Passing Interface," www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.
[22] C. home page, "www.co-array.org."
[23] S. Goudy, S. S. Huang, and Z. Wen, "Translating a high level PGAS program into the intermediate language BEC," Sandia National Laboratories, Tech. Rep. SAND2006-0422, 2006.
[24] J. L. Brown and Z. Wen, "PRAM C: A new parallel programming environment for fine-grained and coarse-grained parallelism," Sandia National Laboratories, Tech. Rep. SAND2004-6171, 2004.
[25] HPC Challenge, "Random Access Rules," http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/.