SAND2008-2155C

Initial Experiences with the BEC Parallel Programming Environment

Mike Heroux, Zhaofang Wen
Sandia National Labs
Albuquerque, NM 87185
{maherou, zwen}@sandia.gov

Junfeng Wu, Yuesheng Xu
Syracuse University
Syracuse, NY 13244-1150
{juwu, yxu06}@syr.edu

Abstract

Bundle-Exchange-Compute (BEC) is a new virtual shared memory parallel programming environment for distributed-memory machines. Different from and complementary to other Global Address Space (GAS) programming model research efforts, BEC has built-in efficient support for unstructured applications that inherently require high-volume random fine-grained communication, such as parallel graph algorithms, sparse-matrices, and large-scale physics simulations.

In BEC, the global view of shared data structures enables ease of algorithm design and programming; and for good application performance, fine-grained (random) accesses to shared data are automatically and dynamically bundled together for coarse-grained message-passing. BEC frees the users from explicit management of data distribution, locality, and communication. Therefore, BEC is much easier to program than MPI, while achieving comparable application performance. This paper presents some initial BEC applications, which show that simple BEC programs can match very complex and highly optimized MPI codes.

1 Introduction

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, often integrated together in a programming environment.) 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 [1] and SHMEM [12] have made an impact on a subset of platforms and applications. Shared memory parallel models such as POSIX Threads [2] and OpenMP [3] 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 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.

A practical programming model must balance 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 nontechnical barriers to acceptance of new programming models, and probably the most important one is the migration of heavily-invested and actively-used legacy (MPI) applications and their expert programmers, along with the costs of development and adoption of new models to a new programming model.

Despite MPI's success, there are still many classes of application for which MPI is not suited. It is also questionable whether MPI will be suitable for future architectures.

1.1 State of the Art

The DARPA HPCS program is in its Phase III (2006-2010) with a focus on programming models [9]. GAS 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 [11] and Cray's Chapel [14]), all to offer GAS as a subset. GAS models can be realized in libraries and language extensions. Examples of GAS libraries are MPI-2 [15], and Cray's SHMEM [12]. Existing GAS language extensions include Unified Parallel C (UPC) [4], Co-Array Fortran (CAF) [5], and the Java-based Titanium [7]. UPC is currently available on many large machines. However, there are still significant challenges on the path to true adoption of GAS models by the parallel programming community. These challenges include the current GAS models' lack of built-in support for a unstructured applications, and the lack of efforts to help smooth transition of MPI legacy applications and their programmers.

1.2 Overview of the BEC Model and Programming Environment

BEC is a parallel programming model and environment. BEC can be used alone; it can be used as an enhancement to an existing environment such as MPI; and it can also function as an intermediate language [8] to other GAS languages such as PRAM C [6] and UPC [4]. Furthermore, it can serve as a bridge between programming models such as virtual shared memory and message passing.

As a parallel programming model, BEC supports virtual shared memory (a.k.a. Global Address Space, or GAS) programming. The global data view of virtual shared memory allows for ease of parallel algorithm design and expression of parallelism, especially random fine-grained parallelism. For efficient implementation, BEC **automatically and dynamically bundles up** fine-grained messages of any data types and arbitrary sizes for coarse-grained communications. It provides programmers with a simple mechanism to minimize the overhead of fine-grained communication without having to manage the data distribution, locality, or communication. The strength of BEC is most apparent for 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 provides users with an easy-to-use programming language extension (to ANSI C) and a light-weight runtime library. The BEC programming language extends ANSI C (ISO C 99) to support parallel programming. The BEC runtime library consists of a small set of functions to let users program in BEC 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 the runtime library directly may be able to write 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 includes one or more Bundle-Exchange-Compute phases as follows. [1]

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 bundle object for multiple bundle-exchange-compute phases.

2. **Exchange:** A call to the 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 calling the *BEC_exchange()* function. 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

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

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(0);
  /* The ``0" means normal exchange  */
  /* mode will be used.              */
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.

1.3 Language Extension and BEC Runtime Library

BEC supports parallel programming in virtual shared memory. The BEC programming environment includes a minor language extension (to ANSI C). For the language extension, the keyword "shared" is added for declaration of globally (logically) shared variables. On a distributed memory machine, a shared variable is physically distributed over multiple processors. With the BEC language extension, shared arrays can be operated on using standard array syntax, such as array indexing (e.g. A[5] for shared array for A).

The BEC programming environment also includes a runtime library, BEC lib. This library can be used with or without the BEC language extension. Even without the BEC language extension, it is still possible to declare and used shared variables, except that array syntax is not allowed for shared arrays and that shared arrays will be manipulated using the BEC lib functions. BEC Lib has a very simple API with about 15 functions altogether. For details, please refer to [16].

The BEC programming environment has a BEC to C translator that converts a BEC program into a program written in C and BEC lib function calls.

1.4 Execution Model and Flow of Control

A BEC program follows the SPMD (Single-Program-Multiple-Data) model: programs are executed asynchronously except at *BEC_exchange* functions. Also, one copy of BEC lib runs on each and every physical processor. The virtual shared memory of BEC is managed by BEC lib, which bundles read and write requests according to physical processor destinations. BEC lib calls upon the message-passing layer (e.g. MPI or Portals) to perform actual transport of data request bundles. When the shared data requests are satisfied, in the case of read requests, the requested data are sent back to the requesting processors. BEC lib maintains hash tables to store remotely fetched shared data, which can be retrieved efficiently for use in local computation.

2 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 sparse linear system solver using the Conjugate Gradient (CG) method, and
- a graph coloring algorithm based on the Largest Degree First (LDF) heuristic.

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.

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 GAS 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 GAS 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 GAS programming abstraction, and advances the state of the art. BEC can help smooth migration from MPI to GAS programming models.

2.1 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 with 9,660 compute nodes. There is one AMD Opteron 2.6 GHZ dual-core on each node, so there are 19,320 compute processor cores in total. More detail about Franklin can be found at http://www.nersc.gov/nusers/resources/franklin/.

2.2 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 [10]; 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 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.

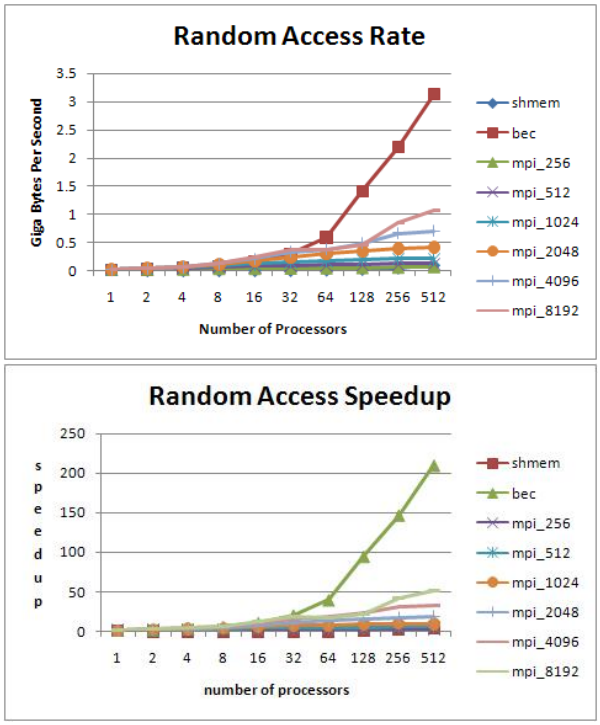

Figure 1. Performance of Random Access

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.

2.3 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. With 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.

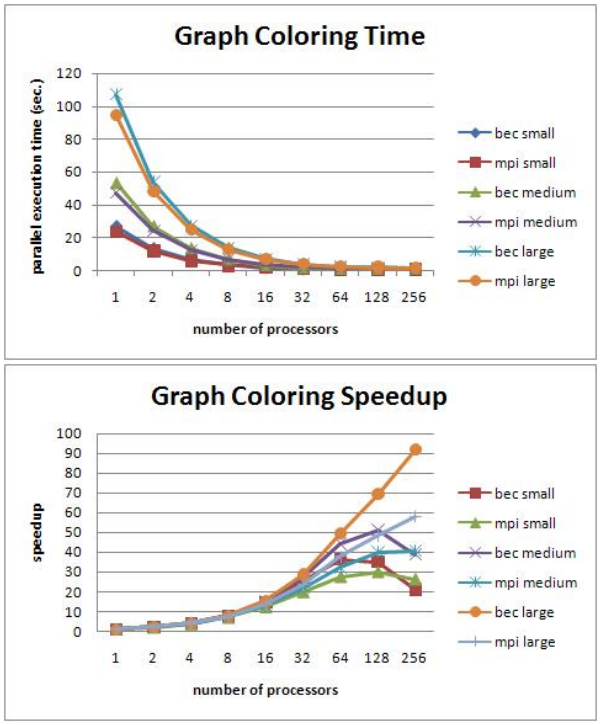

Figure 2. Performance of Graph Coloring

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 implementation are comparable. The BEC implementation started out a little bit slower on low processor count. This is because the BEC bundling (e.g. "BEC_request(...)") operations involve function calls to the BEC lib and associated function call overheads, while in the MPI implementation the manual bundling operations do not involve such function calls and overhead. The performance of the BEC implementation catches up relative to the MPI implementation, because the BEC runtime library has sophisticated built-in data structures (such as hash tables) that are highly optimized (especially as the processor count increases), while the MPI implementation uses general purpose hash tables from the C++ Standard Template Library (STL). In Figure 2, BEC seems to scale better as the number of processors increases; this is because the BEC implementation started out slower. The BEC and MPI implementations have similar performance scaling trend for the same input data set. When the scaling trend turns, the BEC curves trend down more dramatically than the MPI curves; and this is because the MPI message is optimized for the specific application, while the BEC message as prepared by the BEC lib is general. For high processor count, the BEC speedup numbers look better because the BEC implementation started out slower than the MPI implementation for a single processor.

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

Table 1. Code Sizes of the Graph Coloring Programs

Table 1 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 **the MPI implementation uses the C++ Standard Template Library (STL)** for its hashing capabilities. Without calling a third party library, 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.

2.4 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 2. 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 2. Code Sizes of the CG Programs

3 Conclusion

We have presented the BEC programming environment. Some applications have been implemented in

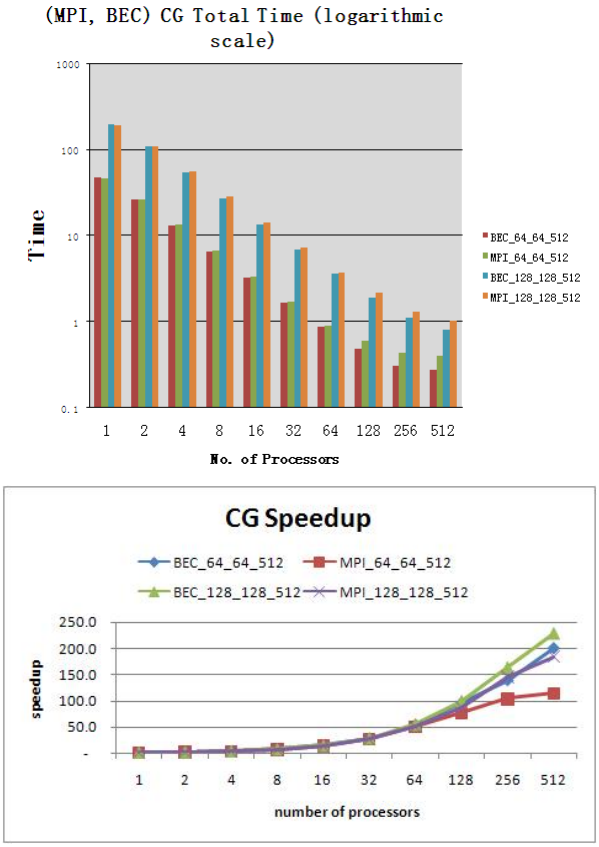

Figure 3. Performance of Conjugate Gradient

BEC based on a Beta release. 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 global address space (GAS) 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 GAS 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 GAS programming abstraction, and advances the state of the art. BEC can help smooth migration from MPI to GAS programming models.

4 Acknowlegement

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.

Sue Goudy, Jonathan Brown, and Shan Shan Huang 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] www.csm.ornl.gov/pvm/pvm_home.html.

[2] www.opengroup.org/certification/posix-home.html.

[3] OpenMP fortran application interface version 1.1. www.openmp.org.

[4] UPC language specification (v 1.2). http://www.gwu.edu/ upc/documentation.html.

[5] www.co-array.org/.

[6] 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 Lab, 2004.

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

[8] Sue Goudy, Shan Shan Huang, and Zhaofang Wen. Translating a high level PGAS program into the intermediate language BEC. Technical Report SAND2006-0422, Sandia National Lab, 2006.

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

[10] HPC Challenge. Random Access Rules. http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/.

[11] IBM. The X10 Programming Language. http://x10.sourceforge.net/.

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

[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] Cray (url). Chapel — The Cascade High-Productivity Language. http://chapel.cs.washington.edu/.

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

[16] Zhaofang Wen, Junfeng Wu, and Yuesheng Xu. Bec specification and programming reference. Technical Report SAND2007-7617, Sandia National Lab, Albuquerque, NM, 2007.