SANDIA REPORT

SAND2004-3485
Unlimited Release
Printed July 2004

Advanced Parallel Programming Models Research and Development Opportunities

Ronald B. Brightwell and 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-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2004-xxxx
Unlimited Release
Printed July 2004

Advanced Parallel Programming Models Research and Development Opportunities

Ron Brightwell, Zhaofang Wen

Abstract

There is currently a large research and development effort within the high-performance computing community on advanced parallel programming models. This research can potentially have an impact on parallel applications, system software, and computing architectures in the next several years. Given Sandia's expertise and unique perspective in these areas, particularly on very large-scale systems, there are many areas in which Sandia can contribute to this effort. This technical report provides a survey of past and present parallel programming model research projects and provides a detailed description of the Partitioned Global Address Space (PGAS) programming model. The PGAS model may offer several improvements over the traditional distributed memory message passing model, which is the dominant model currently being used at Sandia. This technical report discusses these potential benefits and outlines specific areas where Sandia's expertise could contribute to current research activities. In particular, we describe several projects in the areas of high-performance networking, operating systems and parallel runtime systems, compilers, application development, and performance evaluation.

Acknowledgment

We would like to express our appreciation to the people who helped us during the preparation of this technical report. Sue Goudy reviewed early drafts and identified several important Sandia applications. Thomas Christopher provided background on earlier parallel programming models and suggested the idea of a possible implementation of UPC on the PIM architecture. Mike Heroux shared his invaluable experiences about suitable applications for UPC and directed us to important applications, such as the Trilinos project (the Epetra package) and Co-array Fortran applications, that may be suitable for this work. Neil Pundit reviewed several drafts and provided input and guidance. Jonathan L. Brown assisted with editing and revising this report, provided several UPC coding examples, and conveyed early experiences with the language. Last but not least, Bill Camp provided many insightful comments and constructive suggestions, which helped to give a more complete and practical perspective to this report.

Contents

Advanced Parallel Programming Models Research and Development Opportunities

1 Introduction

There is currently a large research and development effort within the high-performance computing community on advanced parallel programming models. The Center for Programming Models for Scalable Parallel Computing [40], led by Argonne National Laboratory, includes researchers from several other national laboratories and universities. The purpose of this center is to coordinate research and development activities on current and advanced parallel programming models. These activities can potentially have an impact on parallel applications, system software, and computing architectures. Given Sandia's expertise and unique perspective in these areas, particularly on very large-scale systems, there are many areas in which Sandia can contribute to this effort.

In particular, much research is being conducted into Partitioned Global Address Space (PGAS) programming models. PGAS models have characteristics of both distributed shared memory models and distributed memory message passing models. Research is being conducted on library-based and language-based implementations of PGAS models. In this report, we describe two PGAS languages that are extensions of C (Unified Parallel C) and Fortran (Co-Array Fortran). Based on our initial analysis, we propose possible research and development activities that would benefit Sandia as well as other organizations engaged in the overall parallel programming models research effort.

2 Motivations

There are several motivations for this initial analysis of possible areas of research and development on parallel programming models and PGAS models in particular. First, PGAS models are not completely unfamiliar. The Cray T3 series of machines supported a PGAS programming model via the SHMEM [11] programming interface and an early implementation of the UPC compiler. Several Sandia researchers have had direct experience with application development and system software development on these platforms. The Cray T3E is widely accepted as a highly scalable and effective parallel platform, and the programming model and architectural support for it, played an important part in its success.

In fact, the hardware support for distributed memory shared memory on the T3 influenced the design of the network interface for ASCI Red.

A key characteristic of the PGAS model is the reliance on extremely low latency, low overhead data movement. This a good match for Sandia's hardware architecture and system software approach. PGAS implementations offer data movement operations similar to those required by MPI, but with significantly less overhead. The combination of a lightweight compute node kernel [37] and the Portals [3] data movement layer appear to support PGAS implementations extremely well. Because these are unique to Sandia, the parallel programming model research community is not currently considering or targeting this architecture. It is Sandia's interest to insure that PGAS programming models do not mandate features that would negatively impact their ability to work in this environment.

There are currently several important DOE applications, for example those developed at PNNL using Global Arrays, that rely on efficient PGAS model implementations. Sandia's Red Storm machine could be an important platform for the entire DOE complex. It would be very beneficial to other labs to have PGAS models work well on the Red Storm machine. There is also significant interest in Red Storm from the National Security Agency (NSA), currently the primary funder of UPC research efforts.

Additionally, Sandia is also currently involved in advanced architecture research projects that could potentially provide the basis for a future large-scale machine. It is not clear that current programming models will continue to be effective and efficient on these architectures.

It is desirable to have a qualitative and quantitative understanding of the PGAS model for Sandia applications. The PGAS model offers performance and ease-of-use benefits, and it would be advantageous to understand the characteristics of our applications that make them amenable to a PGAS implementation.

Finally, parallel programming models research was specifically identified as a weakness by the 9200 external review committee in 2002 and 2003.

In the rest of this report, we survey the current state of parallel programming model research efforts and the models developed over the past two decades. We then provide a detailed discussion of the state of technology of PGAS model implementations. Finally, we propose possible programming model research activities applicable to Sandia.

3 Parallel Programming Models

Developing applications for tera-scale systems is difficult, and typical parallel applications achieve only a small fraction of peak performance, especially at scale. Reasons for this can include cumbersome programming models, lack of support for component software, mismatch between application structure and programming model, and mismatch between programming model and machine architecture.

In general, it is hard to express the inherent parallelism of the algorithms in a parallel program. It is also difficult to efficiently map the parallelism of a program to the hardware because of the wide variety of parallel hardware architectures and rapid advances in computing technologies. Until recently, no single parallel programming model has satisfied the often conflicting requirements of efficiency, portability, and expressiveness necessary to develop high performance applications for rapidly evolving parallel systems. In the following section, we provide brief summaries of different models and emphasize each model's strengths and weaknesses.

3.1 SIMD Model

The Single Instruction, Multiple Data (SIMD), or data parallel, model is characterized by a single thread of execution. Different data items are manipulated with the same execution instructions. Conditional statements are used to mask the input and output of the data items in an operation. The Connection Machine CM-2 is the canonical example of a SIMD machine. It supported data parallel languages, such as C* [10] and CM Fortran. High-Performance Fortran [5] and ZPL [7] are other more recent examples of data parallel languages. While these languages were easy to program and understand, the lack of independent branching limited the types of applications that could run effectively.

3.2 Message Passing Model

In the message passing model, a set of cooperating sequential or parallel processes, each with its own local address space, interact through explicit data movement function calls, or some higher-level abstraction. The programmer has complete control over the distribution and decomposition of the data. This model is generally portable to all architectures, since explicit data movement is an innate part of parallel programming. Message passing programs have also proved to scale extremely well. However, overhead is significant for small messages, and this model is generally perceived to be more difficult to program. This

model has been characterized as the “assembly language level programming” of parallel computing. The canonical example of this model is MPI.

3.3 Shared Memory Model

The shared memory model is characterized by a single address space that can be accessed by concurrent threads of execution. This model is generally perceived to be easy to program to, since physically remote memory can be accessed via assignment statements and expressions. However, manipulation of shared data requires explicit synchronization, and there is no exploitation of data locality. It is difficult to port to scalable systems with processors that do not share physical memory. This model has demonstrated poor scalability on large-scale shared memory systems, and the cost of large shared memory systems is typically much greater than distributed memory systems. Examples of this model are POSIX threads and OpenMP compiler directives.

3.4 Hybrid Models

There are also hybrid models, sometimes called mixed-mode programming, that combine two different models into the same program. This model is typically used on clusters of shared memory machines, where a shared memory model is used within a node and a message passing model is used to exchange data between nodes. Examples of hybrid models include OpenMP with MPI [23] and HPF with MPI [20].

Using a shared memory model on a node avoids the overhead and data replication from using explicit message passing, but at the cost of greater complexity. Mixed-mode programming does not guarantee the benefits of the two models being used. In some cases, a mixed-mode code can run slower than a single-mode code. For example, multiple threads within an OpenMP process may have to synchronize on MPI calls, eliminating the opportunity to overlap computation and communication. It may also be non-trivial to break single-level parallelism in MPI codes into multi-level parallelism needed for a hybrid approach.

3.5 Partitioned Global Address Space Model

The partitioned global address space (PGAS), or distributed shared memory model, provides the programmer with a shared memory model abstraction that can be efficiently im-

plemented on distributed memory systems. In this model, threads share a common global address space that is partitioned among the threads, and each partition has affinity to a single thread. Data locality is managed explicitly at the application level. This model is generally perceived to be easier to program and understand, compared to explicit message passing. Many view the move to this model from message passing as analogous to the move from assembly language to Fortran. There are some drawbacks to this model. As in the shared memory model, explicit synchronization is required for access to shared data. There are library and language implementations of this model. Library implementations include SHMEM and MPI-2 one-sided communications. Language implementations include Unified Parallel C, Co-Array Fortran, and Titanium. We provide a more detailed overview of PGAS model implementations below.

3.6 Less Popular Models

The programming models described above constitute a large percentage of those that are in use today. Other programming models have been developed by the research community. We outline these in this section for completeness.

The PRAM is a SIMD shared-memory model [38]. It was widely used in the 1980's and early 1990's to devise parallel algorithms (mostly non-numerical). This model was favored by academic algorithm designers for its simplicity, which allowed them to concentrate on discovering the inherent parallelism of problems. The shortcoming of this model is that communication cost is not accounted for, and, as a result, most of the algorithms developed with PRAM do not have efficient implementations on today's real world parallel machines. In summary, this is a theoretical model that did not work well in practice.

The Bulk-Synchronous Parallel (BSP) model was introduced by Valiant in 1990 [42, 21]. It abstracts the characteristics of a parallel machine into three numerical parameters p, g, and L, corresponding to processors, bandwidth, and periodicity, respectively. The model differentiates memory that is local to a processor from that which is not, but, for the sake of universality, does not differentiate network proximity. The BSP model supports shared memory or PRAM-style algorithms. It emphasizes the viability of an alternative direct style of programming where, for the sake of efficiency, the programmer retains control of memory allocation. It was shown that optimality to within a multiplicative factor close to one can be achieved for the problems of Gauss-Jordan elimination and sorting by transportable algorithms that can be applied for a wide range of values of the parameters p, g, and L. While these algorithms are fairly simple themselves, descriptions of their behavior in terms of these parameters are somewhat complicated.

The **Wavefront Arrays** model was proposed in 1982. This model supports the data flow model on a systolic array, which is suitable for VLSI implementation [22, 24]. System logic is in use in many ASIC designs.

The **Macro data flow** model was exemplified in the language SISAL [29, 28]. SISAL is a functional parallel programming language developed in the early 1990's. It combines modern language features with readable syntax and mathematically sound semantic foundations to provide an easy vehicle for parallel programming. Its optimizing compiler and runtime support system software provide portability, high performance, and deterministic execution behavior. It is available on all Unix-based uniprocessor and shared memory multiprocessor systems, and developmental versions exist for several distributed memory systems as well. In practice, it was not easy to achieve high performance with large-scale parallelism using this model.

The Message Driven Computing model was developed in the early 1990's [8]. It is a style of computing with two unique characteristics. First, messages, not sequential processes, convey both control and data. Indeed, there are no sequential processes unless they are programmed explicitly in terms of messages. Secondly, computation is invoked by the presence of a collection of messages at the same location. This model did not achieve widespread adoption or use.

4 PGAS Programming Model Implementations

In this section, we provide a more detailed discussion of current implementations of the PGAS programming model. These implementations can be divided into library-based and language-based. Library-based implementations typically consist of a set of callable function routines from C, C++, or Fortran. These libraries usually do not require any unique support from compilers, operating systems, or parallel runtime systems. On the other hand, language-based implementations are extensions of standard programming languages, and depend on compiler or translator support and may require support from a parallel runtime system.

4.1 Library-Based Implementations

Global Arrays (GA) [26, 27] provides a shared memory view of distributed data structures for a MIMD parallel program. It allows each process to access logical blocks of physically distributed multidimensional arrays as if they were located in shared memory. The GA

programming model is primarily a memory rather than interprocess communication model, as it provides interfaces for data movement between shared and local memory. [25]. GA was developed at Pacific Northwest National Laboratory (PNNL), and is the basis for the NWChem application, as well as several other codes at PNNL. GA is built upon a low-level message passing system called the Aggregate Remote Memory Copy Interface (ARMCI).

The SHMEM [11] library is a one-sided programming model developed by Cray for the T3 series of machines [36] circa 1994. It is a set of communication primitives based on simple underlying get/put functionality, with some additional shared memory-like primitives, such as remote fetch-and-increment and remote atomic swap operations. The SHMEM implementation on the Cray T3 leveraged the hardware and runtime system support for globally addressable memory. The SHMEM library on the T3 had both C and Fortran interfaces, and was used as the underlying data movement layer for implementations of UPC and Co-Array Fortran. After SGI acquired Cray, it offered a SHMEM implementation for its shared memory platforms as well. Recently, Ames Laboratory has developed a portable implementation of a subset of the SHMEM API called Generalized Portable SHMEM, or GPSHMEM [30]. This implementation runs on a wide variety of machines and clusters.

The MPI 2.1 standard [31] defines an interface and semantics for one-sided communication operations. MPI allows a process to expose a portion of its address space as a "window" that other processes can manipulate. One-sided operations in MPI are more heavyweight than typical one-sided interfaces, mostly due to the need to allow for portability and to support heterogeneous computing.

4.2 Language Extensions

Unified Parallel C (UPC) [19] and Co-Array Fortran [32] are the two most popular PGAS language extensions. We describe them in more detail below.

Titanium is a language that supports the SPMD model of computation in which processes communicate through a global address space [16]. It gives users access to object-oriented technology and lets users write explicitly parallel code to express the parallelism in the algorithms. It adds primitives for SPMD synchronization, layout and access to shared data structures, efficient multidimensional arrays, immutable classes, and region-based memory management. An optimizing compiler is associated with the language. The compiler performs special analysis and optimizations peculiar to shared memory parallel programming languages. Such capabilities are lacking in commercial compilers.

OpenMP [34] is a set of directives and runtime library routines that was developed by a consortium of vendors to enable portable shared-memory parallel programming. OpenMP

directives specify how a program's computations are to be distributed among the executing threads at run time.

4.3 Hybrid Models

Languages and libraries can potentially complement each other. One such example is the combination of OpenMP and MPI, where OpenMP is used to control multiple threads in a single node and message passing with MPI is used for communication among nodes. [18].

5 Unified Parallel C

Unified Parallel C [19] is a relatively new programming language. Since its first technical report in 1999, it has received much attention from the government, universities, and private sectors [9, 1, 17].

Unified Parallel C (UPC) was designed as a unification of three earlier parallel C languages.

AC: Designed and developed at Center for Computing Sciences, originally for the CM-5 from Thinking Machines and later adapted to the Cray T3 series [6].

Split C: Designed and developed by at the University of California-Berkeley [15].

PCP: Designed and developed at Lawrence Livermore National Laboratories [4].

UPC is not a superset of these three predecessors. Rather, it is a distillation of the essential features needed to support programming for a wide variety of parallel computer architectures in an efficient manner. UPC maintains the C philosophy of keeping the language concise, expressive, and by giving the programmer the power of getting closer to the hardware.

UPC is designed to support the Distributed Shared Memory Model, which is in between the Distributed Memory Model and the Shared Memory Model. The hope is to achieve the balance between the ease of use (of the shared memory model) and the ability to exploit data locality (of the message passing model). The idea behind UPC is that users should be able to view the underlying machine model as a collection of threads operating in a common global address space [19]. Specifically, each thread can access data resident in:

- the local part of the address space
- the shared part of the address space with affinity to that thread
- the shared part of the address space with affinity to other threads

There is no explicit message passing, as UPC features are at a significantly higher-level than MPI.

5.1 The Current State of UPC

The following organizations are actively involved in UPC research and development:

- **Academia**: UC Berkeley (funded by NSA, DOE), George Washington Univ. (funded by NSA), George Mason Univ. (funded by DOD), Michigan Tech (sponsored by HP-Compaq)
- **Government**: ARSC, IDA, LBNL, LLNL, NSA, DOD, DOE
- **Vendors**: Cray, CSC, HP-Compaq, Etnus, IBM, Intrepid Technology, SGI, SUN
- **Supercomputing Centers Users**: European Center of Parallelism of Barcelona, the Sweden National Supercomputer Center, etc.

5.2 UPC Compiler implementations

- GCC-based compiler for SGI Intrepid

  Implemented as a C language dialect translator, in a fashion similar to the implementation of the GNU Objective C compiler, the GCC UPC compiler extends the capabilities of the GNU GCC compiler. This is supported on SGI workstations and servers running the MIPS instruction set, the IRIX (release 6.5 or higher) OS, and the mips2 32-bit ABI. Supported systems include Origin 2000 super-servers and Octane workstations. By default, this release of the GCC UPC compiler supports systems with as many as 256 independent processing units. (For more details, see `http://www.intrepid.com/upc/index.html`.)

- **GCC based compiler for Cray T3E**: The first UPC compiler.

- **Compaq UPC compiler V2.0 (for AlphaServer/Quadrics, MPI + C compiler)**: Compaq UPC is a fully-conforming implementation of the UPC language, with some extensions, primarily for compatibility with the Compaq C and Compaq C++ products. Compaq UPC supports the UPC language specification V1.0 developed by the UPC Consortium and released in February 2001. Compaq UPC Version 2.0 for Compaq Tru64 UNIX introduces support for SMP systems such as the Compaq Alphaserver ES Series. (For more details, see [9].)

- **MuPC V1.0 UPC Translator and Run Time System**: (by Michigan Tech) MuPC is a run time system for UPC. It enables UPC programs to run on a variety of platforms (Tru64, Linux clusters, Solaris). The compilation script *mupcc* first invokes a third-party UPC-to-C translator to translate the user's UPC program into C. References to shared data and other UPC constructs are translated into MuPC run time system calls. MuPC uses Pthreads and MPI to implement a UPC run time system interface specified by HP.

  The MuPC system is a portable, open source reference implementation of UPC. MuPC was not designed for high performance. In V1.0 each remote reference in a UPC program has about twice the latency of an elementary MPI message passing operation. MuPC is intended as a convenient system for learning UPC and experimenting with its features. It is not intended for production work. Work is currently underway to improve MuPC performance (see `http://www.upc.mtu.edu/MuPCdistribution/`).

- **The UPC compiler by LBNL** (Beta release, 2003): The main targets are the IBM SP platform and PC clusters. It is implemented as a portable compiler infrastructure (UPC$\rightarrow$C) with optimization of communication and global pointers. The runtime system is to be shared by multiple compilers (UPC, Titanium, and Co-Array Fortran). The compiler is based on the Open64 compiler for C originally developed at SGI. It has an IA-64 back-end with some ongoing development. New compiler optimizations are being identified and implemented. (Note: Open64 is a suite of optimizing compiler development tools for Intel Itanium(TM) systems running Linux. It is available on SourceForge.net.) The portable communication layer GASNet for Unified Parallel C is designed to run on multiple transport layers:

  - UNIX System V shared memory
  - IBM SP (LAPI)
  - Myrinet (GM)
  - Quadrics (Elan3)
  - Scalable Coherent Interface

  - Virtual Interface Architecture
  - Infiniband
  - MPI

  (For more details, see `http://upc.nersc.gov/`.)

- **IBM UPC compilers** (under development): UPC is part of the BG/L project [1].
- **Sun UPC compiler** (Released December 2001): No details available.

5.3 Performance Studies

There have been some initial performance studies of UPC on small clusters. There are no available studies on large-scale platforms.

In [14], performance of UPC and MPI programs were compared on a Compaq AlphaServer SC cluster using the Compaq UPC 1.7 Compiler. The AlphaServer SC is based on the AlphaServer ES40, which is an SMP machine with four Alpha EV67 processors. The interconnect is a Quadrics switch with 440Mb/s throughput (206Mb/s under MPI) and 6 μseclatency. Experiments showed that UPC compares unfavorably with MPI in implementing coarse-grain algorithm since there are no built-in collective routines. The user is forced to express collective communication operations at the UPC language syntax level, while in the case of MPI, collective operations are separate library calls that can be optimized by vendors. In these tests, the performance difference between the UPC and MPI programs was roughly equal to the time difference between the UPC code and the MPI code executing the collective operations. The authors concluded that the availability of an optimized collective communication library is crucial to the performance of UPC. The UPC consortium is currently engaged in defining and implementing a collective communication interface for UPC that can be optimized at low levels.

In [2], performance of UPC versus a hybrid system, MPI with OpenMP, was studied. The test platform was a 64-node Compaq AlphaServer SC with four 667 MHz Alpha EV67 processors and 2GB of memory per node running Tru64 Unix version 5.1.) Experiments showed that the UPC fine-grain algorithm in a distributed environment did not perform as well as OpenMP. However, a coarse-grain algorithm in UPC had similar performance for simple programs. But, as communication became more complicated, MPI achieved better performance due to the well-tuned collective communication library routines. In a shared memory environment, UPC could achieve similar performance to OpenMP and MPI, but currently does not. UPC is as simple to use as OpenMP and the critical subset of MPI, and simpler than the full MPI API, but lacks the flexibility of OpenMP in thread control.

5.4 Overview of UPC Language Features

The following section describes some of the UPC language features.

In this example, A, B, and C are declared as two-dimensional shared arrays, whose elements are distributed across the threads by row-sized blocks for A and C and by element-sized blocks for B in a round-robin fashion. In this case, whole rows of A and C have affinity to the same thread.

upc_forall is a parallel loop. The fourth field in the loop is called "affinity". In this example, it indicates that the thread which has row $C[i]$ should execute the i^{th} iteration. Private pointers are added as an optimization.

- **Memory**: Private memory can be accessed only by one thread. Shared memory can be accessed by all threads. Each shared memory partition has affinity to one thread (and can be accessed faster by that thread)

- **Shared memory**: Shared variable declarations can be qualified as *relaxed* or *strict*. Strict shared variable implies serialization semantics for accesses to the variable. All accesses to a strict shared variable are sequence points. Relaxed shared variable implies weak semantics for accesses to that variable.

- **Parallel model**: There is one global copy of each shared variable. There is one copy per thread of each private variable. Each thread executes code in the SPMD fashion.

- **Pointers in UPC**
  Pointer declarations in UPC are very similar to those in C. There are four distinct possibilities: private pointers pointing to the private space, private pointers pointing to the shared space, shared pointers pointing to the shared space, and shared pointers pointing to the private space. For example,

```
int *A;    /* private to private */
shared int *B;    /* private to shared  */
int * shared C;   /* shared to private  */
shared int * shared D;  /* shared to shared  */
```

  In the most flexible case, private pointers declared in one thread can point to private space in another thread.

```
shared [N] int  A[M][N];
shared     int  B[N][P];
shared [P] int  C[M][P];
           int  local_B[N][P];
           int *local_A_row
           int *local_C_row;
           int  i,j,k;

    ...

    /* multiply in parallel with c's row affinity */
    upc_forall (i = 0; i < M; i++; &C[i]) {

        local_C_row = (int *) &C[i][0];
        local_A_row = (int *) &A[i][0];
        for (j = 0; j < P; j++) {
            local_C_row[j] = 0;
            for (k = 0; k < N; k++) {
                local_C_row[j] += local_A_row[k] * local_B[k][j];
            }
        }
    }

    upc_barrier;
```

Figure 1. UPC Code Example

- **Data Distribution**: Shared objects (and pointers to shared objects) are created by a declaration with the shared type qualifier. Shared objects cannot be dynamic (they cannot be on-stack), and they cannot be a component of a structure. Pointers to shared objects can be dynamic or part of a structure. Non-array objects have affinity to thread 0.

- **Example**: Array objects are allocated to threads in a round-robin fashion. For example, given the following declaration:

```
shared [2] int x[2][3]
```

  The elements (column-major in memory) will have affinity to the threads as follows:

```
thread 0: x[0][0], x[1][1];
thread 1: x[1][0], x[2][0];
thread 2: x[0][1], x[2][2]
```

  It is possible to specify the block size:

```
shared [2] int x[2][3]
```

  The elements will have affinity to the threads as follows:

```
thread 0: x[0][0], x[1][0];
thread 1: x[0][1], x[1][1];
thread 2: x[0][2], x[1][2]
```

- **THREADS** is a constant at runtime. It can either be specified as a compilation constant in a static environment, or it can be specified at load time in a dynamic environment. THREADS can be used in the declaration of a shared array. For example,

```
shared [2] int x[THREADS][2]
```

The elements will have affinity to the threads as follows.

```
thread 0: x[0][0], x[0][1];
thread 1: x[1][0], x[1][1];
thread 2: x[2][0], x[2][1]
```

- **UPC parallel loop construct**:
  upc_forall(expr; expr; expr; affinity) statement
  Here *affinity* specifies which thread executes which iteration. Note that:
  - there is no language support for dynamic scheduling
  - there is no communication or synchronization between iterations
  - there is no nesting of "upc_forall" loops
  - there is no parallel reduction. (Note: this issue will be addressed by the newly-released UPC Collective Operation Specification [13].)
- **Thread Synchronization** in UPC is facilitated through *locks* and *barriers*. Critical codes section can be executed in a mutually exclusive fashion using locks. The related library calls are *upc_lock*, *upc_unlock*, and *upc_lock_attempt*. A barrier is a point in the code at which all threads must arrive, before any of them can proceed any further. UPC provides two versions of barrier synchronization: the regular barrier – *upc_barrier* – and the split-phase barrier – *upc_notify; ...; upc_wait;*.

5.5 Pros and Cons of UPC

Pros

- Extension of familiar syntax of C
- Locality exploitation: blocking, affinity
- Implementations on several platforms
- Incremental code parallelization

- One-sided communication can avoid the overhead associated with two-sided message passing. From a performance standpoint, one-sided data movement eliminates the need for message selection and transmission of message selection information in messages. From a resource utilization perspective, no buffering is required to support unexpected messages or different message mode semantics. MPI also supports one-sided communication semantics, but in a more general way. MPI one-sided operations were intended to allow for implementations that use two-sided mechanisms.

Cons

- In a distributed memory environment, calculating shared array indexes to achieve locality could complicate the code, making it hard to read and debug. Also, a miscalculation on affi nity would mean accessing none-local elements, and thus lead to severe performance degradation.
- Semantic extensions are sometimes unintuitive
- Affi nity mechanism is limited
- Implementations are limited
- Small user community
- Parallel loop construct *upc_forall* is very restrictive.

6 Co-Array Fortran

Co-Array Fortran (CAF) [32] is an extension to Fortran. CAF supports the SPMD model of computation in which a collection of process images execute asynchronously and share data using one-sided communication through an explicit syntax. Process images interact by reading and writing data objects that are marked as co-arrays. The programmer is responsible for synchronization among images using memory barriers and flexible lightweight synchronization primitives that support pair-wise, group, and barrier synchronization. CAF does not require a single (physical) global address space. The effectiveness of the CAF model has been demonstrated in a number of applications ([35, 12, 33]). The Naval Research Lab has developed CAF versions of the NRL Layered and Coastal Ocean Models and achieved performance superior to hand-coded OpenMP and MPI implementations (see `http://www.co-array.org/` for more details). However, the Cray T3E is the only machine that supports CAF.

Co-Array Fortran (CAF) [32] was designed to answer the question 'What is the smallest change required to convert Fortran 95 into a robust, efficient parallel language?'. The answer was a simple syntactic extension to Fortran 95, although work started from Fortran 77. Formerly called F–, CAF is a small set of extensions to Fortran 95 for parallel processing. A complete language definition was published in 1998.

A CAF program is interpreted as if it were replicated a number of times and all copies were executed asynchronously. Each copy is called an **image** with its own set of data objects. The array syntax of Fortran 95 is extended with additional trailing subscripts in square brackets to give a clear and straightforward representation of any access to data that is spread across images.

References without square brackets are to local data, so code that can run independently is uncluttered. Communication is involved only between images where there are squared brackets, or where there is a call to a procedure which contains square brackets.

The programmer is responsible for synchronization among images using memory barriers and flexible lightweight synchronization primitives that support pair-wise, group, and barrier synchronization.

6.1 The Current State of Co-Array Fortran

The effectiveness of the CAF model has been demonstrated in a number of applications ([35, 12, 33]). The Naval Research Lab has developed CAF versions of the NRL Layered and Coastal Ocean Models and achieved performance superior to hand-coded OpenMP and MPI implementations. For more details, see `http://www.co-array.org/`.

The Cray T3E is currently the only machine that supports Co-Array Fortran. This is an important reason why the language has not been used widely. To address this issue, Rice University is developing a portable CAF compiler, and the University of Minnesota is developing a portable runtime system for CAF. Both the compiler and the runtime system will be released in Fall 2003.

6.2 Co-Array Fortran Language Features

Co-Array Fortran is a simple syntactic extension to Fortran 95 that converts it into a robust, efficient parallel language. It has the look-and-feel of Fortran and requires Fortran programmers to learn only a few new rules. These new rules are related to two fundamen-

tal issues that any parallel programming model must resolve: work distribution and data distribution. (See [32] for details.)

- A Co-Array Fortran program is replicated to images with indexes 1, 2, 3, ... and runs asynchronously on them all. The number of images is fixed throughout execution.

- The intrinsic function *num_images()* returns the the number of images, and *this_image()* returns the index of the invoking image.

- Normal array syntax is extended to include square brackets and the resulting objects are called co-array sub-objects. For example,

```
REAL, DIMENSION(N)[*] :: X,Y
X(:) = Y(:)[Q]
```

  declares that each image has two real arrays of size N. If Q has the same value on each image, the effect of the assignment statement is that each image copies the array Y from image Q and makes a local copy in array X.

- A reference to a co-array without square brackets is a reference to a local object.

- The intrinsic functions *sync_all(/wait/)* , *sync_file(unit)*, *sync_memory()*, and *sync_team(team /,wait/)* are barriers to synchronize the executing image with another image, a set of other images, or all other images.

- The intrinsic functions *start_critical()* and *end_critical()* can be used to mark critical code regions to prevent race conditions. (Each image holds an integer called its critical count. On entry, the count for the image shall be positive. On exit, the count is decremented by one.)

- I/O is performed independently by each image. However, there is a single file system and a single set of units shared by all images. The new keyword TEAM is used to identify which images may perform I/O on a given unit. (*sync_file(unit)* is an intrinsic for marking the progress of input-output on a unit.

- Execution of a STOP on any image causes all images to cease execution.

- Simple examples:

```
X       = Y[PE]  ! get from Y[PE]
Y[PE]   = X      ! put into Y[PE]
Y[:]    = X      ! broadcast X
Y[LIST] = X      ! broadcast X over subset of PE's in array LIST
Z(:)    = Y[:]   ! collect all Y
S = MINVAL(Y[:]) ! min (reduce) all Y
B(1:M)[1:N] = S  ! S scalar, promoted to array of shape (1:M,1:N)
```

There are some advantages of CAF over MPI in terms of ease of use [32]. For example, co-array syntax requires no include files, no status buffers, no model initialization, no message tags, no error codes, no library-specific variables, and no variable size information. Moving an arbitrarily complicated Fortran 90 data structure creates no problem for co-array syntax, but there is no simple equivalent in MPI.

7 Possible Areas of Parallel Programming Model R&D at Sandia

Parallel programming model research needs support from many areas, including programming languages, compilers, runtime systems, operating systems, network, I/O, applications, application library development, and so forth. In this section, we propose some of the possible R&D activities in these areas, in particular, those closely related to Sandia.

7.1 PGAS Support for Red Storm

The ASCI Red Storm machine is a joint project between Sandia and Cray, Inc. to deliver a 40 teraFLOPS machine in late Summer of 2004. This machine will run a lightweight compute node operating system developed by Sandia (called Catamount), and the custom network interface will support the Portals 3.3 data movement interface developed by Sandia. Sandia's previous ASCI machine, ASCI Red, currently runs the previous generation of these two components, the Cougar lightweight kernel and Portals 2.0. We believe that the hardware and software architecture of Red and Red Storm are ideal for supporting a PGAS programming model such as UPC. We would propose a research and development project to implement and evaluate UPC on Red and Red Storm. Initially we would use the ASCI Red platform since development hardware is available. The similarity in the software

environment between the machines would allow us to identify limitations that could impact the development of the software for Red Storm. This would help to insure that Red Storm could support an efficient and scalable implementation of UPC. We would also investigate issues with respect to the specific hardware for Red Storm, such as compiler support for the AMD Opteron.

7.2 Applications, Performance Study, and Technology Comparison

As a feasibility study, we will implement some of Sandia's applications using the PGAS model. We have identified the Trilinos Project – in particular a component called Epetra – as one such application, and we will collaborate with Mike Heroux on this study. The Trilinos Project is an effort to develop parallel solver algorithms and libraries within an object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific applications. Its basic building blocks include operations such as the following.

- Dot product

```
for (i=1; i <= n; i++)
sum = A[i]*B[i]
```

- Vector update

```
for (i=1; i <= n; i++)
B[i] = x*A[i]+y*B[i]
```

- Solving of a linear system of equations

```
Ax = B
```

  given a matrix *A* and a vector *B*, where *A* is typically a sparse matrix.

We will implement these operations using UPC. Since the Trilinos project is mainly implemented in C++, we will use UPC to implement the internals of the classes while keeping the class interfaces. This way the UPC implementation can integrate with the rest of the libraries.

For technology comparison, we will then compare the performance of the UPC implementations with the MPI implementation. We will also look at the ease of programming using UPC as compared to MPI.

We will conduct a similar study on some Sandia applications using Co-array Fortran. Suitable applications are being identified.

7.3 Compilers

We have several choices depending on which platforms we choose to support UPC. There are also some practical issues.

- The MuPC project (Michigan Tech) is using a UPC compiler front-end from EDG (Edison Design Group, a for-profit organization). Their runtime is free. They are also willing to give us the binary of this compiler front-end. But we will not be able to modify it (both technically and legally).

- An optimizing UPC compiler is available from Berkeley. They are very interested in helping us out. Their UPC compiler runs on the IBM machine at NERSC, and the beta release is available. It is a portable compiler that also runs on Myrinet clusters, the HP/AlphaServer machines, and (although not with the best performance) on top of MPI. There are some issues related to usability of the compiler which they are trying to iron out before a release, and performance tuning as well as high level optimizations will be ongoing as part of their compiler effort.

- Commercial UPC compiler from HP-Compaq. [1]

- For Co-Array Fortran, a new portable compiler and a new portable runtime system are being developed at Rice University and the University of Minnesota, respectively. Both will be available in September 2003. They are interested in helping us out.

[1] To use the UPC compiler and runtime, HP charges roughly $3750 for a 2-processor machine and up to $8000 for a 1024-processor machine.

Besides porting, the performance of these compilers will be analyzed. In the process of implementing applications and performance study, new compiler optimization opportunities will be identified. We will collaborate with the compiler vendors to have our required optimizations implemented, and, whenever possible, we will develop new compiler optimization techniques ourselves.

7.4 Runtime Systems

Similarly with the Co-array Fortran compilers, different runtime systems are provided with the different versions of the UPC compilers.

- The MuPC project (Michigan Tech) has a portable runtime system which will run on any system that supports MPI and Pthreads. The downside is that this runtime system is not optimized.
- UC Berkeley has developed a lightweight communication and run-time layer, the GASNet, for global address space programming languages. The GASNet library currently runs over Quadrics/elan, Myrinet/GM, and IBM LAPI, and/or any MPI 1.1 implementation. Besides UPC, GASNet is also being used as the networking layer for the Titanium language (a high-performance parallel dialect of Java).
- The HP-Compaq UPC also comes with a runtime library.
- For Co-Array Fortran, a new portable runtime system is being developed at the University of Minnesota. It will be available in September 2003.

The runtime systems that come with the compilers from the above sources may not give us the best possible application performance on Sandia's computational platforms. Therefore, these runtime systems may have to be modified, and in some cases we may need to develop new runtime libraries. For example, it may be necessary for us to re-implement some of the libraries based on Sandia's Portal layer.

Two specifications on UPC Parallel I/O API [39] and UPC Collective Operations [13] have recently been released. Corresponding libraries will become available later. These libraries, even if portable to Sandia's environment, may not provide the best possible performance, which again will require re-implementation to suit Sandia's specific environment. For example, if we choose to port UPC to Red Storm, the UPC Parallel I/O library may need to be re-implemented based on Red Storm's (Parallel I/O) libraries.

7.5 Practical Language Extensions

7.5.1 Linking Programs in Different Languages

A large percentage of Sandia's scientific programs are written in Fortran. It is important to be able to link code written in UPC with code in Fortran. Similarly, it is also important to be able to link code written in CAF with code in C.

Solutions depend on which compilers we use. For example, if we use a compiler which translates UPC code into $C + MPI$ code, then the linking of UPC code with Fortran code becomes a problem of linking C code with Fortran code, which we already know how to do. In the case that the UPC compiler is not a preprocessor, much work will be needed to resolve this issue. If the runtime system for UPC is not MPI, we will research how to resolve this linking issue. Similar issues exist for linking CAF code with C code.

7.5.2 UPC++

Many of Sandia's applications and libraries are written in C++. It might be natural if the language were UPC++ rather than UPC. (Some consideration was given to this idea in the UPC community, but no real progress has been made.) One simple possibility is to extend the C++ language with a similar set of parallel programming constructs as the one in UPC from C. A possible implementation would be to implement a preprocessor that translates UPC++ into MPI & C++. It is still an open problem whether an optimizing compiler for UPC++ can be developed similar to the UPC optimizing compiler.

7.6 Developing PGAS Application Libraries and Framework

To make it easier for application developers to adopt the PGAS model, the next step after the initial feasibility study would be to develop libraries and framework to support application programming in the PGAS model. (These are different from the runtime library.) These libraries will include the ones to support specific types of applications, and specifically they will also include libraries to complement the UPC and CAF languages in a way similar to the libraries complementing the C language.

7.7 PGAS Models and Data Services

Sandia applications are becoming increasingly more dependent on a data services for advanced I/O and visualization functionality. The Red Storm parallel runtime system is currently being enhanced to support running a parallel job that spans both compute and service partitions in order to provide data services more efficiently. This distribution of services is rather straightforward to support within an MPMD programming model such as provided by MPI. It is not clear how to efficiently support such services from within a PGAS programming model like UPC. The tightly coupled, global address space abstraction may provide opportunities for optimizing the data path between advanced I/O and visualization services. We believe that this is an important area that needs to be investigated in order to support our applications within a PGAS framework.

7.8 Programming Model Research for Future Supercomputer Architectures

Sandia currently sponsors several research projects on advanced architectures for the supercomputers of 2010 and beyond, such as the PIM (Processing in Memory). At this point PIM seems to be the most promising architecture that we will need to use for the supercomputers in 2010. The current programming model used at Sandia (large processors + MPI) does not use the processors on the PIMs effectively. It is necessary for us to investigate alternative programming models. Certain characteristics of the PIMs make a programming model like UPC suitable. For example, PIMs provide huge number of small threads, and UPC can be implemented using *parcels* [41], a very light weight communication mechanism. Research on new programming models is necessary in order to support future generations of supercomputers.

8 Conclusion

We have surveyed the past and the current state of parallel programming model research. We propose to introduce the PGAS model, and we have identified related research activities that are applicable to Sandia. We also propose to research programming models for Sandia's future supercomputer architecture. These efforts would allow Sandia to keep up with the state of the art in the constantly advancing field of parallel programming model research. Briefly, the proposed projects include UPC for Red Storm, application performance analysis, compiler and runtime support for Sandia's system software environment,

language extensions for UPC, application data services support, and PGAS model support on advanced architectures.

References

[1] C. Cascaval R. Barik. Unifi ed Parallel C on BlueGene/L, 2002. presentation at BG/L Tahoe workshop.

[2] K. Berlin. UPC vs. MPI and OpenMP: Analysis of a hiybrd approach to parallel programming, 2002.

[3] R. B. Brightwell, W. Lawry, A. B. Maccabe, and R. E. Riesen. Portals 3.0: Protocol building blocks for low overhead communication. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters*, Ft. Lauderdale, Florida, April 2002. 2002 International Parallel and Distributed Processing Symposium.

[4] K. Warren E. Brooks. Development and evaluation of an effi cient parallel programming methodology, spanning uniprocessor, symmetric shared-memory multiprocessor, and distributed-memory massively parallel architectures. 1995.

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

[6] J. M. Draper W. W. Carlson. Distributed data access in AC. pages 39–47, 1995.

[7] Bradford L. Chamberlain, Sung-Eun Choi, E. Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. ZPL: A machine independent programming language for parallel computers. *IEEE Transactions on Software Engineering*, 26(3):197–211, March 2000.

[8] Thomas W. Christopher. Early experience with object-oriented message driven computing. October 1990.

[9] Compaq Computer Coporation. *Compaq UPC Programmer's Guide*, 2002.

[10] Thinking Machine Corporation. *Getting Started with C**, February 1992.

[11] Cray Research, Inc. *SHMEM Technical Note for C, SG-2516 2.3*, October 1994.

[12] R. W. Numrich E. T. Kapke, A. J. Schultz. A parallel class library for Co-Array Fortran. 1999.

[13] E. Wiebel, D. Greenberg, and S. Seidel. UPC Collective Operations Specifi cation V1.0. Technical report, May 2003.

[14] F. Cantonnet T. El-Ghazawi. UPC performance and potential: A NPB experimental study, 2002.

[15] D. E. Culler et al. Parallel programming in Split-C. pages 262–273, 1993.

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

[17] K. Yelick et al. UPC NERSC/LBNL, 2002. Presentation slides.

[18] W. D. Gropp et al. Analyzing the parallel scalability of an implicit unstructured mesh CFD code. pages 395–404, 2000.

[19] W. W. Carlson et al. Introduction the UPC and language specifi cation. Technical Report CCS-TR-99-157, 1999.

[20] Ian Foster, David R. Kohr, Jr., Rakesh Krishnaiyer, and Alok Choudhary. A library-based approach to task parallelism in a data-parallel language. *Journal of Parallel and Distributed Computing*, 45(2):148–158, September 1997.

[21] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of ACM*, 22(2):251–267, 1994.

[22] H. T. Kung and C. E. Leiserson. Systolic arrays (for VLSI). Technical Report CS-79-103, Carnegie Mellon University, 1978.

[23] Y. H. He. Hybrid openMP and MPI programs on the SP: Successes, failures, and results, 2003.

[24] H.T. Kung, K. S. Arun, R. J. Gal-Ezer, D. B. Rao. Wavefront array processor: Language, architecture and applications. *IEEE transactions on computers*, (c-31):1054–1066, 1982.

[25] I. Foster J. Neplocha, R. J. Harrison. Explicit management of memory hierarchy. pages 185–200, 1997.

[26] R. J. Littlefi eld J. Neplocha, R. J. Harrison. Global arrys: A portable shared memory programming model for distibuted memory computers. pages 340–349, 1994.

[27] R. J. Littlefi eld J. Neplocha, R. J. Harrison. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:197–220, 1996.

[28] J. R. Gurd, C. C. Kirkham, A. P. W. Bhm. The manchester dataflow computing system. *Experimental Parallel Computing Architecture*, December 1987.

[29] J.T. Feo, D.C. Cann, R. R. Oldehoeft. A report on the sisal language project. *Journal of Parallel and Distributed Computing*, 10:349–365, December 1990.

[30] R. A. Kendall K. Parzyszek, J. Neiplocha. A generalized portable SHMEM library for high performance computing. pages 401–406, 2000.

[31] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[32] J. K. Reid R. W. Numrich. Co-array fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, 1998.

[33] R. W. Numrich. Object-oriented LU decomposition with Co-Array Fortran. 2000.

[34] OpenMP Architecture Review Board. OpenMP Fortran Application Interface Version 1.1. *http://www.openmp.org*, 1999.

[35] K. Kim R. W. Numrich, J. K. Reid. Writing a multigrid solver using Co-Array Fortran. pages 390–399, 1998.

[36] J. C. Peterson R. W. Numrich, P. L. Springer. Measurement of communication rates on the Cray T3D interproc essor network. April 1994.

[37] P. L. Shuler, C. Jong, R. E. Riesen, D. van Dresser, A. B. Maccabe, L. A. Fisk, and T. M. Stallcup. The Puma operating system for massively parallel computers. In *Proceedings of the 1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group, 1995.

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

[39] T. El-Ghazawi, F. Contonnet, P. Saha, R. Thakur, R. Ross, and D. Bonachea. UPC-IO: A Parallel I/O API for UPC. Technical report, May 2003.

[40] The Center for Programming Models for Scalable Parallel Programming. `http://www.pmodels.org`.

[41] Thomas L. Sterling and Hans P. Zima. Gilgamesh: A Multithreaded Processor-In-Memory Architecture for Petaflos Computing. 2002.

[42] L. G. Valiant. A bridging model for parallel computation. *Communication of ACM*, 33(8):103–111, 1990.

DISTRIBUTION:

1 MS 0321
Bill Camp, 9200

1 MS 1110
David Womble, 9210

1 MS 0370
Scott Mitchell, 9211

1 MS 0310
Mark D. Rintoul, 9212

1 MS 1111
Bruce Hendrickson, 9214

1 MS 1110
Suzanne Rountree, 9215

1 MS 0318
Jennifer Nelson, 9216

1 MS 0321
Robert Leland, 9220

1 MS 1110
Jim Tomkins, 9220

1 MS 1110
Neil Pundit, 9223

1 MS 1110
Ron Brightwell, 9223

1 MS 1110
Jonathan Brown, 9223

1 MS 1110
Zhaofang Wen, 9223

1 MS 0817
Jim Ang, 9224

1 MS 0376
Ted Blacker, 9226

1 MS 0822
David White, 9227

1 MS 0316
Paul Yarrington, 9230

1 MS 0378
Randy Summers, 9231

1 MS 0378
Patrick Chavez, 9232

1 MS 0316
Sudip Dosanjh, 9233

1 MS 0310
John Aidun, 9235

1 MS 9018
Central Technical Files, 8945-1

2 MS 0899
Technical Library, 9616