Procedia Computer Science

Volume 29, 2014, Pages 1491–1503

ICCS 2014. 14th International Conference on Computational Science

Supporting relative debugging for large-scale UPC programs

Minh Ngoc Dinh[1*], David Abramson[1], Jin Chao[1]
Luiz DeRose[2], Bob Moench[2], Andrew Gontarek[2]
[1]*The University of Queensland, St Lucia, Australia.*
[2]*Cray Inc., St Paul, USA.*
{m.dinh1, david.abramson, c.jin}@uq.edu.au; {ldr, rwm, andrewg}@cray.com

Abstract
Relative debugging is a useful technique for locating errors that emerge from porting existing code to new programming language or to new computing platform. Recent attention on the UPC programming language has resulted in a number of conventional parallel programs, for example MPI programs, being ported to UPC. This paper gives an overview on the data distribution concepts used in UPC and establishes the challenges in supporting relative debugging technique for UPC programs that run on large supercomputers. The proposed solution is implemented on an existing parallel relative debugger CCDB, and the performance is evaluated on a Cray XE6 system with 16,348 cores.

Keywords: UPC, PGAS, relative debugging, data decomposition

1 Introduction

The advent of programming languages over the years has improved not only the efficiency in developing large-scale applications, but also the performance of those programs that run on supercomputing platforms. Since the introduction of the Message Passing Interface (MPI) [1], several other parallel programming languages and environments [2] allow programmers to seamlessly and easily control/synchronize large number of processing threads, and address large memory space that is distributed across those threads. Lately, Unified Parallel C (UPC) [3] has caught the attention of the HPC community due to its ease to code and its performance promise. As a result, a number of traditional parallel programs, for example C/MPI programs, have been ported to UPC. However, while the similarity between the two languages helps reducing the number of errors in the porting process, verifying the correctness of the new UPC program against the existing MPI code remains a challenging task. First, many large-scale programs manipulate enormous multi-dimensional data

* Corresponding author. Tel.: +61 7 3195 1645
E-mail address: m.dinh1@uq.edu.au

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2014

doi:10.1016/j.procs.2014.05.135

structures. Second, those complex structures are distributed across thousands of processes (because no single machine can accommodate them).

This paper gives an overview of UPC, especially the data distribution concept under the PGAS programming model [4], and establishes the challenges in testing and debugging a UPC program. In particular, the paper discusses the differences between several data-parallel programming models, from ZPL [5] to MPI and finally PGAS. In our previous works, we discussed solutions for both ZPL and MPI [1]. This paper describes an extension to support PGAS programming model, especially the UPC programming language. Further, through performance measurements, we show that our debugging framework can display, compute and compare large runtime datasets in parallel and provide reasonably low latency for online usage.

The paper is structured as follows. In section 2, we present the background motivation. First, we introduce UPC, with a focus on how UPC runtime distributes and manages the global shared variables. The discussion provides the motivation and the foundation for the data-framework proposed in section 3. Second, we discuss recent efforts in supporting debugging of UPC programs; highlighting the lack of support in handling UPC global shared objects. In section 3, we introduce the Cray comparative parallel debugger called *CCDB*, and discuss how CCDB is suitable to handle partitioned global address space at scale. Section 4 closely examines how global view of runtime state is reconstructed and presents the implementation details. Section 5 delivers the analysis and evaluation of the performance obtained on a Cray XE6 system with 16,384 cores. We conclude the paper in section 6 with some discussion about future enhancements.

2 UPC parallel programming language

Unified Parallel C (UPC) combines several distributed/shared-memory C languages including AC, Split-C, and Parallel C Preprocessor (PCP) [3] to provide an alternative to MPI. While following a distributed-memory programming model (SPMD), UPC is designed to leverage the ease of programming using the shared-memory paradigm, especially, enabling the exploitation of data locality. As a result, a UPC user is presented with a global shared, partitioned address space, where variables may be directly read and written by any processor, but each portion of the variable might be physically associated with a single processor.

To achieve that, UPC implements the Partitioned Global Address Space (PGAS) model [4]. This programming model supports direct access to all data from all computing threads. Further, by assigning an *affinity* for a particular process to portions of the shared memory space, PGAS exploits locality of reference to improve the global-shared memory access performance [4]. To further improve the performance, UPC runtime incorporates constructs that allow placing data near the threads that manipulate them to minimize remote accesses. The UPC-PGAS model is evaluated using three application benchmarks (BT, LU, and SP) from the NPB suite [6] to examine the performance impact of data affinity and data access on both shared-memory and cluster-based parallel systems. The authors find the UPC implementations of the NPB benchmarks produce comparative performance against the MPI versions.

2.1 Data distribution in UPC

In UPC, a variable can be either a *shared* object or a *private* object. A private object has its copies in the private space of each thread while a shared variable has the first instance of the object assigned to thread 0, and all computing threads access to the same memory space. With shared array objects, UPC distributes the array blocks across the different threads. More importantly, to exploit inherent data locality in application, UPC defines the "physical" mapping between shared array elements and UPC threads. This association is called *affinity*, and it indicates which thread "owns" a particular data

element. UPC programmers can define such affinity in the code in order to keep the shared data that will be regularly processed by a given thread (and rarely accessed by others) associated with that thread. The general declaration for a shared array with affinity is shown below:

```
shared [blocking-factor] <data_type> array_name [number-of-elements]…[]
```

When the programmer does not specify a blocking-factor, UPC automatically assumes a blocking-factor of 1. Consider the following UPC snippet:

```
shared int x[13];       // shared array with blocking-factor of 1
shared [3] int y[13];   // shared array with blocking-factor of 3
```

thread 0	x[0] x[3] x[6] x[9] x[12]
thread 1	x[1] x[4] x[7] x[10] x[13]
thread 2	x[2] x[5] x[8] x[11]

shared int x[13]
(a) Default blocking factor

thread 0	y[0] y[1] y[2] y[9] y[10] y[11]
thread 1	y[3] y[4] y[5] y[12] y[13]
thread 2	y[6] y[7] y[8]

shared [3] int y[13]
(b) User-defined blocking factor

Figure 1: Examples of data decomposition in UPC

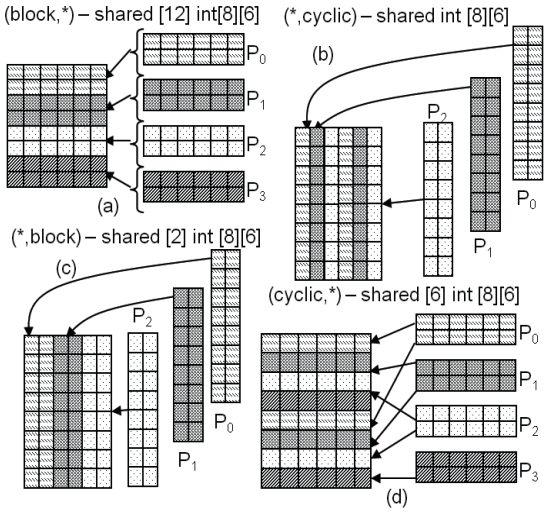

Figure 2: Example of block/cyclic decomposition in UPC

For the shared array *x*, UPC uses a default blocking-factor of 1 and distributes the elements in a round robin fashion (i.e. cyclic) as shown in Figure 1a. Alternatively, with shared variable y, UPC distributes the array with a blocking-factor of 3, as shown in Figure 1b. Furthermore, UPC defines two constants: "THREADS" is the total number of computing threads, and "MYTHREAD" allows a thread to identify itself. While the affinity concept seems simple and easy to use, it is powerful enough to let users describe different type of domain decompositions (such as block/cyclic decompositions) as employed by HPF [2] and MPI [1]. For example, given the 8x6 array of integers in Figure 2, a user can specify different types of decomposition in UPC. Importantly, UPC treats a multi-dimensional array as a 1D array and vectorizes it before distributing the elements across the threads. This is an important feature and it simplifies the algorithm used to reconstruct global arrays, as discussed later in section 4.2.2.

2.2 Debugging UPC applications

TotalView [7] debugger can display UPC shared objects by fetching the right data portion from the UPC thread with which it has an affinity. For instance, with shared scalar objects, TotalView obtains runtime data from thread 0. With shared arrays, TotalView displays both the thread ID along with the pointer to the data segments that belongs to that particular thread. In other words, similar to supporting MPI programs, data from a distributed array is displayed rankwise. Furthermore, TotalView

understands pointer-to-shared data, as well as the target of the pointer to shared variables. The Allinea Distributed Debugging Tool (DDT) [8], also supports regular debugging tasks for UPC codes. However, in general, it lacks the supports for exploring UPC global shared-arrays and variables. Other general debugging tasks such as capturing and displaying back-trace (or stack traceback) can be done with GDB-UPC [9]. GDB-UPC supports UPC shared type qualifier for shared address space as well as collective UPC mode of operation for breakpoints and single code stepping.

With Berkeley UPC [10], a program can also be (partially) debugged using a regular C debugger. In particular, one or more computing threads can be attached to a regular C debugger at various points during execution. Because this process requires the translation of UPC code to regular C code, a full debugging support is not available. For example, the debugger will show the C code emitted by the translator, rather than the original UPC code. Nevertheless, important debugging information such as stack traces is still provided.

Other typical debugging methods are also supported for UPC programs. For instance, UPC-SPIN [11] provides a framework for the model-checking of the inter-thread synchronization functionality. By transforming the UPC synchronization primitives to equivalent code snippets in the SPIN's modeling language, this framework generates and verifies models of UPC programs to detect thread-synchronization issues such as data/memory race and deadlocks. Another example is UPC-TRACE [12], which is useful for generating trace files and local memory reports.

While a suite of different tools as presented above might provide enough support for debugging UPC code of small scale, the lack of support for exploring large datasets, which are distributed under the PGAS model, limits the verification and debugging of UPC programs that run on large supercomputers. In the next section, we discuss how CCDB – a Cray comparative debugger, is suitable to handle partitioned global address space at scale.

3 Supporting data distribution for PGAS programming model

CCDB is a Cray parallel debugger, which implements the *relative debugging* paradigm. Relative debugging was first introduced by Abramson et al. [13] in 1995 as an automated debugging technique. It supports the process of porting existing sequential and vector codes to new platforms. The authors also introduced Guard, a relative debugger, and presented several case studies where newly ported programs can be compared against an existing code to locate defects. Watson and Abramson enhanced Guard to cater for the relative debugging process of parallel programs including ZPL codes and C/MPI programs [14-18]. Importantly, Guard embeds and implements several data-mapping algebras so that key data structures, which are distributed across multiple processes, can be combined and compared centrally. Recently, both Guard and its data-mapping technique have been enhanced to support large-scale applications, which run across tens of thousands of processing cores [19]. Consequently, Guard's mechanism for providing global view of the program's state has evolved significantly over the years. CCDB was developed on top of Guard and inherits both the relative debugging methodology as well as the supports for exploring and processing large distributed data objects.

First, consider supporting ZPL codes. Because ZPL is a global-view parallel language [5], a programmer writes code without concern about which processors that will execute it. Therefore, while debugging ZPL code, the user should just query for the data structures disregard of their decomposition and distribution. Since ZPL's runtime system maintains a description of each parallel array and provides information such as the array's rank, structure dimension's size and information on the array's distribution across the processes, CCDB retrieves these metadata to determine the location and bounds of the block that resides in the process, and provides an algebra to map each element of the partitioned arrays back to the global array. Such algebra is substantial in restructuring small arrays, but shows severe performance issue when the size of the structure approaches terabytes.

Second, consider supporting MPI codes. In contrast to ZPL as a global-view language, MPI library is a local-view parallel framework. That means there is no concept of global address space and MPI runtime treats runtime data as process's private objects. As a result, the programmer must perform the data distribution across processes manually, and he/she is capable of describing the data decomposition pattern. Importantly, since data decomposition is a burden that is placed on the local-view programmer, regular block/cyclic decomposition schemes becomes general practice in coding with MPI. Because there is no data distribution information provided by the runtime system, to support MPI code, CCDB provides a special function, called *blockmap*, which is inspired by the *block-cyclic data distribution* scheme [20]. It includes a collection of expressions that describe the decomposition scheme. The full semantics of blockmap function can be found in other publication [19]. With *blockmaping algebra*, global array can be reconstructed by mapping *blocks* of elements from partitioned arrays instead of each individual element. Such algebra significantly improve the performance in reconstructing very large arrays that are distributed across many computing processes [19]. Further, blockmap function is useful in constructing the global view of the state of scientific codes at runtime because it can handle details such as ghost-band (e.g. halo cells) and the effect of boundary condition on the sub-structures. Blockmap's user can describe the size of the ghost-band in the blockmap function. Before the sub-structures are mapped back to the global structure, such information is used to define a slicing process that automatically removes the halo cells from each sub-structure.

Finally, comparing to global-view languages such as ZPL, and local-view languages such as C/MPI, local-view languages with global address spaces such as UPC offer several advantages. First, while enforcing programmers to divide the expression of their computation between the processing threads in an SPMD style of programming, it is significantly easier compare to MPI because of the PGAS concepts. Therefore, with a shared array object, a programmer still can query and manipulate a global object without concern of how the array elements are distributed by the runtime system. Second, similar to global-view languages, data distribution information can be retrieved for debugging purposes. Finally, UPC distributes global shared object using a *blocking-factor*, and thus motivates the use of the aforementioned blockmapping algebra to provide sound performance. In section 4, we describe the current blockmapping algebra and discuss the required enhancement in order to support UPC programs.

4 Implementation

4.1 CCDB's architecture

CCDB employs a client/server model, where the debugger is divided into a single front-end client, provides text-based control using Client API and the User Interface; and multiple servers, to ensure that the processes being debugged can be distributed onto multiple processors and can be controlled independently (Figure 3). Network API performs communication between the client and backend servers. This layer interfaces with highly scalable network infrastructure such as MRNet [21]. The client process is attached to the root node of the tree while debug servers are attached to the leaf nodes. The tree layout is explicitly determined when the debugger invokes a parallel program based on the number of processes involved. Descriptions for other components such as Dataflow Compiler/Engine, Statistic API and AIF API can be found in related publication [22].

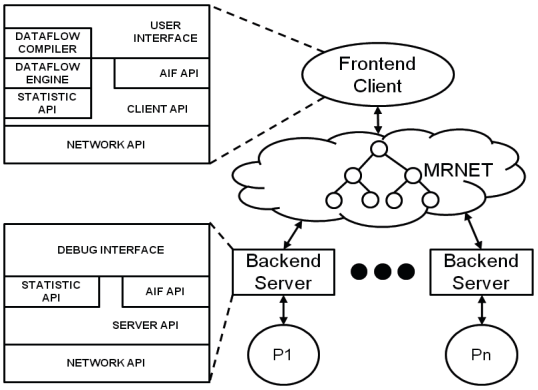

Figure 3. CCDB's client-server architecture

4.2 Global view of UPC-shared arrays

As discussed earlier, the type of data decomposition scheme (i.e. regular block/cyclic decomposition) allowed in UPC is substantially similar to that used in MPI implementation. Therefore, we enhance the blockmapping algebra to handle UPC shared arrays. The following sections explain the current algebra and the details of the improvement.

4.2.1. Blockmapping agebra

To efficiently build the global matrix view given a collection of sub-arrays obtained from remote debug servers, CCDB implements a *blockmapping algebra* that aims to find the biggest block of contiguous elements that can be mapped onto the global matrix. Note that sub-arrays obtained by debug servers retain the layout as to the original global array.

First, we define several data array notions. We denote $P=<p_1,p_2,...,p_n>$ as list of n processes. Let $data(d_1,d_2,...,d_k)$ denotes the shape of an array. In this case, k is the rank of the array and d_i is the size of i^{th} dimension of the array. For example, if $A=data(4,4,6)$, A is an array of rank 3 (3 dimensional array) with 4 layers of matrices, each has 4 rows and 6 columns. Let $blockmap(A,m_1,m_2,...,m_k)$ represents the decomposition function of a global array. For instance, given the 3D array A above then $blockmap(A,2,2,6)$ indicates that the array A is decomposed into 4 sub-arrays of shape $(2,2,6)$. Furthermore, if the array A above can be flattened onto a single rank array A' (i.e. A' is one dimensional array), then the location of each element in the original array A can be mapped to a new location in the single rank array using the formula (1) below. For example, applying the formula

showing that cell A[2][3][3] can be located at index 72 in the flattened array A'. Second, the process of mapping sub-arrays onto the original global array can be done as follows. We flatten the sub-arrays obtained from debug servers and identify the *biggest block* of contiguous elements that can be mapped directly to the flattened global array. For simple cases where a trivial distribution scheme such as *(block,*)* is applied on a 2D array, the biggest block would be the whole sub-array itself. However, with the example global array A illustrated above decomposed with *blockmap(A,2,2,6)*, the biggest block only has a shape of (2,6). This leaves the sub-array to be further partitioned into 2 blocks of (2,6). This desired biggest block is referred to as a *mappable block* and the size of such block is called *block_size* from here onward. Due to formula (1), elements in less significant ranks are arranged first in the flattened array. Hence, a mappable block can be identified by iterating in reverse through the data vector $data(d_1,d_2,\ldots,d_k)$ and vector $blockmap(A,m_1,m_2,\ldots,m_k)$, and block_size can be calculated using formula (2) below. The iteration finishes when d_i is not equal to m_i.

$$A[x_1]\ldots[x_k] \rightarrow A'[y] \text{ where } y = \sum_{i=1}^{k} q_i x_i \text{ and } q_i = \begin{cases} \prod_{j=k+1}^{k} d_j \mid 1 \le i < k \\ 1 \mid i = k \end{cases} \quad (1)$$

$$block\_size = \prod_{i=n}^{m_i \neq d_i} m_i \quad (2)$$

Let us again consider the global array A with *data(4,4,6)* decomposed with *blockmap(A,2,2,6)*, the size of the mappable block will be $m_3*m_2=6*2=12$, and the mappable block has a shape of (2,6). This means that the first 12 cells in each of the sub-array owned by a process p_i can be copied to the global flatten array respectively. Then the second block of 12 elements can be copied next. The process is illustrated in Figure 4 below. Finally, after the flatten global-array is successfully constructed, it can be converted into the original multidimensional global array by restructuring the elements using the formula (1) above in reverse.

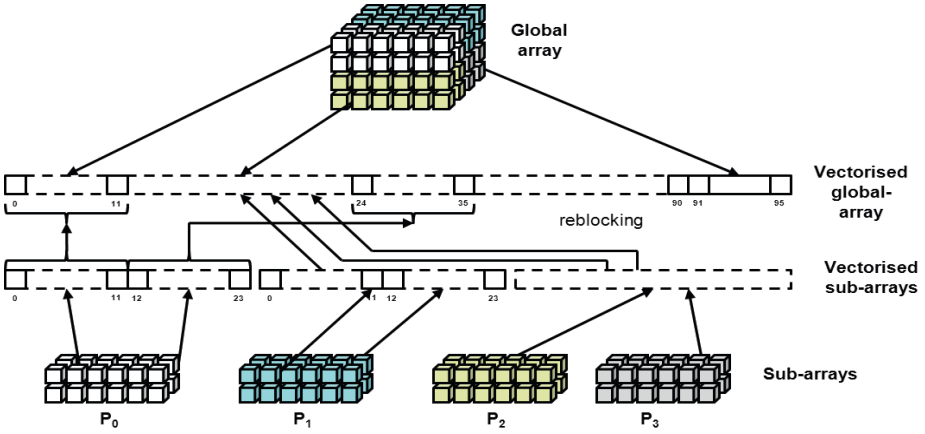

Figure 4. Blockmapping process

4.2.2. Automatic blockmap for UPC shared arrays

While manually describing the decomposition is necessary for MPI implementations, this task can be done automatically for UPC. This functionality requires changes in both debug client and debug server to (1) enable the recognition of UPC shared arrays and obtain the data decomposition information, (2) generate the corresponding blockmap functions, and (3) correctly reconstruct UPC global-shared array with the auto-blockmap function. Doing this allows a range of debugging

commands supported in CCDB such as *print*, *compare*, and *assert*[†] to implicitly work with UPC shared arrays. Below, we present the required enhancement to the blockmaping algebra.

First, because the decomposition of a multi-dimensional shared array in UPC is equivalent to the decomposition of the 1D array given a blocking-factor, an automatic UPC blockmap function treats the original shared array as a flattened 1D array. Second, during the re-blocking process as shown in section 4.2.1, sub-arrays retrieved from debug servers are also flattened. Mappable blocks of elements from these sub-arrays are identified using the given blocking-factor, and they are mapped to the global 1D array in a cyclic manner. Finally, with the global size of the shared array provided by the UPC runtime, the flattened array can be converted into the original multi-dimensional global array in the final step of the blockmapping process.

4.3 Supporting advanced comparison techniques

CCDB significantly improves its performance by using advanced comparison techniques such as a hash-based technique (that reduces the amount of data need to be transmitted and compared) and a direct point-to-point (P2P) technique (that parallelizes the data comparison across a set of debug servers) [19]. Both hash-based and P2P schemes require accessing of two blockmap functions to find the *greatest overlap area* which could be hashed or directly compared by debug servers. We have discussed an algebra to identify such greatest overlapping area in our previous publication [23]. Because the blockmapping algebra is enhanced to support UPC decomposition scheme, techniques such as hash-based and P2P inherently work for UPC programs.

Note that, because the most significant parameter of an automatic blockmap function is the blocking-factor, there is an efficiency issue. While the blockmap overlapping strategy ensures that there is always an overlapping region between two arbitrary blockmap functions, the overlapping region found could be very small in the following cases: (1) the default value for shared array blocking-factor is 1; and (2) user-defined blocking-factors can lead to very small overlapping region – at the limit, only a single cell. In this case, comparison techniques such as hashing and P2P would add no value compared to the traditional comparison scheme. On the other hand, it is unlikely that any parallel program would be efficient if it only allocated a single row or a single column to a given processor, and thus, we can assume that in any real program on any real machine there are likely to be multiple rows or columns assigned to a given processor.

Regarding the current implementation for hash-based scheme in CCDB, while the hashing process is carried out by the debug servers in parallel, constructing the global hashed-signature arrays and comparing them occur sequentially at the client head node. This limits the speed up of the hash-based scheme as shown in section 5.2. This issue is discussed in our previous work publication [23]. However, CCDB allows a user to select which comparison scheme to use. Typically, the hash-based method is first deployed to detect if any data divergence exist, while the P2P method is conducted subsequently to examine a particular region of difference.

5 Performance analysis and evaluation

We analyze the performance of the proposed debugging runtime in handling UPC global-shared arrays, on a Cray XE6 system with 16,384 cores, and UPC codes are compiled with Cray's compiler. Specifically, we closely examine the *print* command and the *assert* command. However, because an *assert* command involves two programs, the tests required two sets of processes and these experiments were run using 8,192 processors (in pairs). In addition, the two programs involve in the test for the

[†] *compare* and *assert* commands allow the comparison of 2 runtime variables. In particular, *compare* command is manually invoked at a breakpoint while *assert* command creates assertions which are verified automatically during the application's execution.

assert command are both UPC programs which perform the same data distribution. We are interested in finding (1) the overhead generated in dealing with UPC runtime and (2) constructing plus executing the auto-generated blockmap functions. Full evaluation of the hash-based and the P2P schemes is published and discussed in [19] where readers can find performance impact on comparing different data distributions. Our experimental setup is similar to one the performance benchmarks presented in [24] for a similar Cray XT3 system which holds a 40,000x40,000 matrix of floats, and distributes it across 20,000 computing processors. Consequently, with 8,192 cores available to us, we generated a >2.5GByte global-structure. Using these parameters, strong scaling experiments are conducted to demonstrate the scalability of the proposed solutions.

For each test, we categorize the time spent in various phases of a particular command into three distinct measurements. First, *"server time"* captures the amount of time spent at the backend for (1) retrieving the variable's type and its private data, (2) performing data comparison tasks such as hashing or P2P tasks, and (3) the time required to transfer the data (hashed signatures or P2P comparison results) to the frontend client. Close examination of these individual elements is presented in our previous publications [19]. Second, *"client time"* displays the time needed to blockmap the data returned from debug servers and display the resulted (global) array to the user. Finally, *"overall time"* simply shows the total elapse time for executing a command.

5.1 Evaluation of the 'print' command

We evaluate the *print* command to understand how the debugger reacts to the increasing amount of data given the same number of debug servers. While TotalView also supports printing UPC global-shared arrays, it only prints sub-arrays along with the rank numbers of the processes. Our print command returns the global-array as declared by the programmer, and this incurs the *blockmapping* overhead. The process to print a global-shared array with CCDB includes several communications between the debug client at the frontend and multiple debug servers at the backend. First, to determine if the variable is actually a shared variable, CCDB requests the variable's type from the debug servers. This information also describes how the UPC runtime decomposes the global structure; thus a blockmap function can be automatically generated. Next, the debug servers acquire the local partitioned arrays, and finally, the debug client generates the global array by applying the auto-generated blockmap function onto the obtained sub-arrays. Because a global array is constructed at the frontend process, there is a constraint on the size of the shared array. As a result, we limit our test to 128 cores, which generates a global structure of ~32Mbytes. Figure 5 present the elapsed time for printing shared arrays with sizes ranging from 32Kbytes to 32Mbytes. The server time increases very little as we double the size of the array and hardly affects the overall performance. However, client time increases linearly. This is due to the blockmap task sequentially processes the increasing amount of data. While this blockmap time dominates the overall printing time when dealing with UPC shared array using auto-generated blockmap functions, we observe a considerable overhead at the client frontend. This overhead is the time required to display the array values on the console to the user; thus it increases linearly with the size of the data structures.

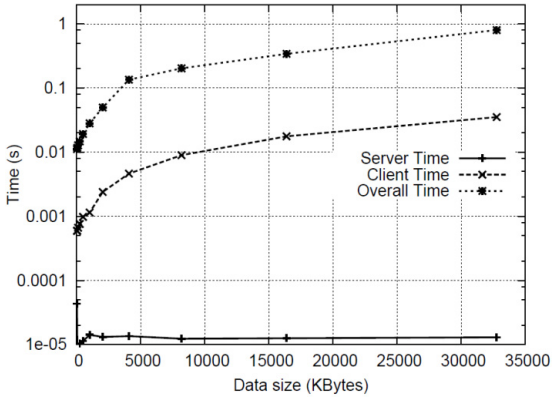

Figure 5. Performance of the "print" command with different data sizes, distributed across 128 cores

5.2 Evaluation of comparative assertions

Comparative debugging mode can be conducted with comparative assertions [19]. As discussed in section 4.3, executing comparative tasks that involve large data structures on large parallel machines requires using either the hash-based comparison scheme or the P2P comparison scheme. Here, we present the strong scaling performance results for both hash-based comparison scheme and P2P comparison scheme. To fully characterize the performance of each approach; various activities that contribute to the overall elapsed time are presented in the following table.

P2P Scheme	Server time = GDB+P2PMap+P2PComm+Comp+ClientComm
	Client time = Blockmap + Result_Display
Hash-based Scheme	Server time = GDB + Hashing + ClientComm
	Client time = Blockmap*2 + Comp + Result_Display

where:

GDB: Time for collecting runtime data from GDB
P2PMap: P2P mapping + preparing sub-blocks.
P2PComm: Communication between debug servers.
ClientComm: Communication time between servers and client.
Hashing: Data hashing time at servers
Comp: Data Comparison time
Blockmap: Data reconstruction time in client.
Result_Display: Time for displaying assertion result.

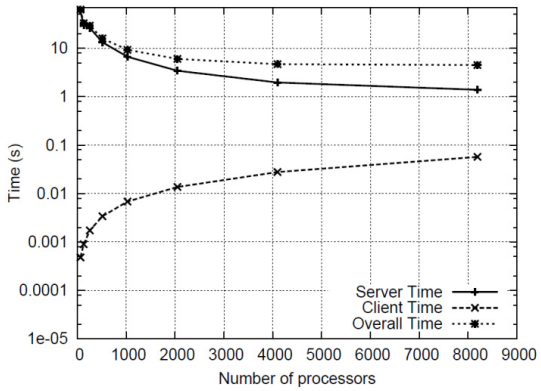

Figure 6. Hash-based comparison - strong scaling results

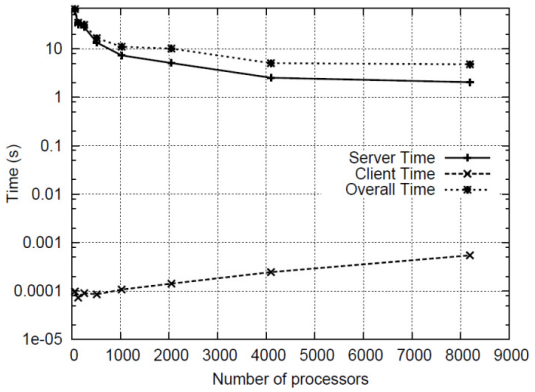

Figure 7. P2P comparison - strong scaling results

Results in Figure 6 and Figure 7 reveal that when the number of cores increases, the server times as well as the overall assertion times for both hash-based scheme and P2P scheme decrease. This is not surprising because more processing resource indicates that more data are processed (hashed or P2P compared) in parallel. However, consider the hash-based scheme results. Because the server time comprises of not only the hashing time by debug servers (Hashing) but also the hashed signatures transfer time (ClientComm), more debug servers implies more data needs to be transferred to the frontend client. It results in the server time decelerates its decrease after around 4,000 cores. Furthermore, the client time also increases proportionally with number of processors. This is because hashing method requires the rebuilding of the global data structures from both programs and also includes the serial comparison time that doubles as the number of processors doubles. These factors limit the speedup for hashing technique after 4K cores, as depicted in Figure 8. On the other hand, at around 4,000 cores, the server time for P2P comparison scheme decreases with lower rate. Two potential reasons can be identified for this observation. First, because P2P communication is current implemented with basic socket protocol, more debug servers try to communicate with each other results in higher network traffic load; thus limits the parallelism gained. Second, more comparison results are collected and transferred to the debug client, which also slow down the overall server time.

These performance results are consistent with the performance data published in our previous work [19]. This observation implies that both the enhanced blockmapping algebra as well as the UPC-supported debugging backend do not introduce substantial overheads.

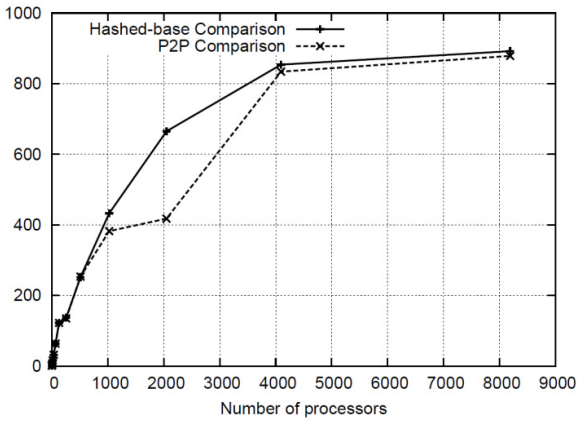

Figure 8. Speedup of overall time against #processors

6 Conclusion and future works

Relative debugging allows the comparison between two executing programs. It enables the detection, and isolation of coding errors that are introduced after a program has been rewritten, enhanced, or has been ported from one programming language or computing platform to another. We have previously demonstrated the applicability of relative debugging for finding errors in various parallel codes ranging from ZPL to MPI implementations. In this paper, we present the enhancement to support data distribution technique implemented by the PGAS programming model, and in particular the UPC programming language. We compare the differences between both ZPL global-view model and MPI local-view model against the global space address model used in UPC, and produce a data-framework that can automatically handle the distributed data for displaying and for comparison purposes. The enhancement enables the use of scalable data comparison schemes such as hash-based and P2P on UPC data structures; thus leverages the debugging process of large-scale UPC applications.

Other than UPC, PGAS programming model is also implemented in some other programming languages such as Titanium [25], Co-Array Fortran [26], and Chapel [27]. For future work, we plan to widen our support to these programming languages and provide a debugging environment in which similar-purpose programs implemented across different languages can be cross-checked and verified against each other using the relative debugging paradigm.

Acknowledgement

This project is supported by the Australian Research Council under the Linkage grant scheme, and is supported by Cray Inc.

References

[1] mpi-forum.org. (2012). *MPI: A Message-Passing Interface Standard*. Available: http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html

[2] H. Richardson, "High Performance Fortran: history, overview and current developments," Thinking Machines Corporation1996.

[3] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and Language Specification," Center for Computing Sciences, Bowie, MD1999.

[4] C. Coarf, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanty*, et al.*, "An Evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, Chicago, Illinois, USA, 2005, pp. 36-47.

[5] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, and L. Snyder, "The high-level parallel language ZPL improves productivity and performance," in *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[6] D. A. Mallón, A. Gómez, J. C. Mouriño, G. L. Taboada, C. Teijeiro, J. Touriño*, et al.*, "UPC performance evaluation on a multicore system," in *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, 2009.

[7] Dolphin Interconnect Solutions Inc. (2007, 03/04/2009). *TotalView Debugger: A comprehensive debugging solution for demanding multi-core applications*. Available: http://www.totalviewtech.com/pdf/TotalViewDebug.pdf

[8] Allinea, "Allinea DDT – A revolution in debugging," ed, 2009.
[9] gccupc.org. (2013, 17/11/2013). *Debugging with GDB UPC*. Available: http://www.gccupc.org/gdb-upc/debugging-with-gdb-upc
[10] D. B. W. Chen, J. Duell, P. Husbands, C. Iancu, K. Yelick, "A Performance Analysis of the Berkeley UPC Compiler " presented at the International Conference on Supercomputing (ICS), San Francisico, CA, USA, 2003.
[11] A. Ebnenasir, "UPC-SPIN: A Framework for the Model Checking of UPC Programs," presented at the Fifth Conference on Partitioned Global Address Space Programming Models, Galveston Island, Texas, USA, 2011.
[12] Berkeley UPC. (2013). *Manual Reference Pages - UPC_TRACE*. Available: http://upc.lbl.gov/docs/user/upc_trace.html
[13] D. Abramson and R. Sosic, "Relative Debugging Using Multiple Program Versions," in *8 th International Symposium on Languages for Intensional Programming*, Sydney, 1995, pp. 3-5.
[14] D. Abramson, R. Sosic, and G. Watson, "Implementation Techniques for a Parallel Relative Debugger," presented at the International Conference on Parallel Architectures and Compilation Techniques, Boston, Massachusetts, USA, 1996.
[15] G. Watson and D. Abramson, "The Architecture of a Parallel Relative Debugger," presented at the 13th International Conference on Parallel and Distributed Computing Systems, 2000.
[16] G. R. Watson, "The Design and Implementation of a Parallel Relative Debugger," Doctor of Philosophy Dotoral, Faculty of Information Technology, Monash University, Melbourne, 2000.
[17] D. Abramson, R. Finkel, D. Kurniawan, V. Kowalenko, and G. Watson, "Parallel Relative Debugging with Dynamic Data Structures," *Communications of the ACM,* vol. 39 pp. 69-77, 1996.
[18] G. Watson and D. Abramson, "Relative Debugging for Parallel Systems," *Proceedings of PCW 97,* pp. 25-26, 1997.
[19] M. N. Dinh, D. Abramson, and C. Jin, "Scalable Relative Debugging," *IEEE Transactions on Parallel and Distributed Systems,* 2013.
[20] A. Petitet. (1995, 20/12/2009). *Block Cyclic Data Distribution*. Available: http://www.netlib.org/utk/papers/scalapack/node8.html
[21] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A Software-based Multicast/Reduction Network for Scalable Tools," in *Proceedings of SC*, Phoenix, AZ, 2003.
[22] M. N. Dinh, D. Abramson, C. Jin, A. Gontarek, B. Moench, and L. DeRose, "A data-centric framework for debugging highly parallel applications," *Software: Practice and Experience,* 2012.
[23] D. Abramson, M. N. Dinh, D. Kurniawan, B. Moench, and L. DeRose, "Data Centric Highly Parallel Debugging," in *ACM International Symposium on High Performance Distributed Computing (HPDC)* Chicago, Illinois, 2010, pp. 119-129.
[24] P. Husbands and K. Yelick, "Multi-Threading and One-Sided Communication in Parallel LU Factorization " in *ACM Supercomputing*, Nevada, USA, 2007, pp. 1-10.
[25] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy*, et al.*, "Titanium: A High-Performance Java Dialect," *Concurrency: Practice and Experience,* vol. 10, pp. 11-13, 1998.
[26] R. W. Numrich and J. Reid, "Co-Array Fortran for Parallel Programming," *ACM SIGPLAN Fortran Forum,* vol. 17, pp. 1-31, 1998.
[27] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel Programmability and the Chapel Language," *International Journal of High Performance Computing Applications,* vol. 21, pp. 291-312, 2007.