A Characterization of Shared Data Access Patterns in UPC Programs

Christopher Barton[1], Călin Caşcaval[2], and José Nelson Amaral[1]

[1] Department of Computing Science,
University of Alberta, Edmonton, Canada
{cbarton,amaral}@cs.ualberta.ca
[2] IBM T.J. Watson Research Center
Yorktown Heights, NY, 10598
cascaval@us.ibm.com

Abstract. The main attraction of Partitioned Global Address Space (PGAS) languages to programmers is the ability to distribute the data to exploit the affinity of threads within shared-memory domains. Thus, PGAS languages, such as Unified Parallel C (UPC), are a promising programming paradigm for emerging parallel machines that employ hierarchical data- and task-parallelism. For example, large systems are built as distributed-shared memory architectures, where multi-core nodes access a local, coherent address space and many such nodes are interconnected in a non-coherent address space to form a high-performance system.
This paper studies the access patterns of shared data in UPC programs. By analyzing the access patterns of shared data in UPC we are able to make three major observations about the characteristics of programs written in a PGAS programming model: (i) there is strong evidence to support the development of automatic identification and automatic privatization of local shared data accesses; (ii) the ability for the programmer to specify how shared data is distributed among the executing threads can result in significant performance improvements; (iii) running UPC programs on a hybrid architecture will significantly increase the opportunities for automatic privatization of local shared data accesses.

1 Introduction

Partitioned Global Address Space (PGAS) programming languages offer an attractive, high-productivity programming model for programming large-scale parallel machines. PGAS languages, such as Unified Parallel C (UPC) [13], combine the simplicity of shared-memory programming with the efficiency of the message-passing paradigm. PGAS languages partition the application's address space into private, shared-local, and shared-remote memory. The latency of shared-remote accesses is typically much larger than that of local, private accesses, especially when the underlying hardware is a distributed-memory machine and remote accesses imply communication over a network.

In PGAS languages, such as UPC, the programmer specifies which data is shared and how it is distributed among all processors. When the data distribution is known at compile time, the compiler can distinguish between local shared data and remote shared

data. This information can be used by the compiler to reduce the time to access shared data [5, 12].

In this paper we report on our experience with a set of existing UPC benchmarks. We start from the premise that understanding data sharing access patterns is crucial to develop high performance parallel programs, especially in a PGAS language. We develop a set of tools to analyze memory behavior, taking advantage of our UPC compiler and runtime system. We characterize the benchmarks with respect to local and remote shared memory accesses, and based on these characteristics we make the following observations:

- Programmers are typically aware of data ownership and make an effort to compute on local data. However, since the data is declared as shared, it will incur the shared memory translation cost, unless it is copied to private memory or dereferenced through private pointers. Requiring programmers to perform both of these actions would increase the complexity of the source code and reduce the programmer's productivity. A more elegant approach is for the compiler to automatically discover which shared accesses are local and to analyze privatize them.
- PGAS languages offer data distribution directives, such as the blocking factor in UPC. Most of the time, programmers think in terms of virtual threads and processors. To develop portable code, programmers do not necessarily select the best distribution for a given platform. Again, there is an opportunity for the compiler to optimize the blocking factor to match the characteristics of the machine. In Section 3 we show several examples in which selecting the blocking factor appropriately, the number of remote accesses is reduced significantly.
- A different way to improve the latency of remote accesses, is to exploit emerging architectures that consist of multi-core chips or clusters of SMP machines – we call these machines hybrid architectures since they are a combination of shared and distributed memory. In this case, a combination of compiler and runtime support can provide an optimal grouping of threads, such that the number of local accesses is increased. In our experiments we estimate the percentage of remote accesses that can be localized.

Several programming models have been proposed for hybrid architecture. Traditionally a combination of OpenMP and MPI has been used to provide efficient communication between nodes while allowing simple distribution of work within each node [9, 21]. However, presenting two very different programming models, shared memory and message passing, to the programmer makes coding of large applications very complex. Beside different data distribution requirements, there are issues of synchronization and load balancing that need to be managed across programming models.

A popular alternative have been Software Distributed Memory Systems (DSMs), such as TreadMarks [2], Nanos DSM [15], ClusterOMP [18], etc. In these systems, the user is presented with a unique programming model – shared memory, and the system takes care of maintaining the coherence between images running on distributed nodes. The coherence is typically maintained at OS page level granularity and different techniques have been developed to reduce the overhead [19]. These characteristics make workloads that have fine-grain sharing accesses and synchronization unsuitable for DSMs [18].

We believe that PGAS languages are inherently more suitable for hybrid architectures, since they are designed to make the user aware of shared data having different latencies. We have previously shown that UPC programs can scale to hundreds of thousands of nodes in a distributed machine [5]. In this work we present evidence that UPC is a suitable language for hybrid architectures, exposing a unique programming model to the user. We argue that a combination of aggressive compiler optimizations and runtime system support can efficiently map a wide range of applications to these emerging platforms.

The remainder of this paper is organized as follows: Section 2 presents an overview of the compiler and runtime system used to collect the results, as well as a description of the benchmarks studied. Section 3 presents the experimental results and discusses what are the issues and opportunities observed. In Section 4 we present the related work and we conclude in Section 5.

2 Environment

In this section we present the environment used for our experiments, and introduce the terminology that we are using throughout the paper. The experiments were conducted on a 32-way eServer pSeries 690 machine with 257280 MB of memory running AIX version 5.2.

2.1 Overview of IBM's Compiler and Runtime System

For this study, we use a development version of the IBM XL UPC compiler and UPC Runtime System (RTS). The compiler consists of a UPC front-end that transforms the UPC-specific constructs into C constructs and calls to the UPC RTS. While the compiler is capable of extensive optimizations, for the purpose of this study we did not enable them. The goal is to observe the sharing patterns in the applications and to gage the possible opportunities for optimizing shared memory accesses.

The RTS contains data structures and functions that are used during the runtime execution of a UPC program, similar to GASNet [7]. In the RTS we use the Shared Variable Directory (SVD) to manage allocation, de-allocation, and access to shared objects. The SVD provides the shared memory translation support and is designed for scalability. Every shared variable in a UPC program has a corresponding entry in the SVD. The compiler translates all accesses to shared variables into the appropriate calls in the RTS to access the values of shared variables using the Shared Variable Directory (SVD). Given that accessing a shared variable through the SVD may incur in several levels of indirection — even when the shared access is local — automatic privatization of local shared accesses by the compiler yields significant performance improvements [5].

2.2 Performance and Environment Monitoring (PEM)

We used the PEM infrastructure [10, 22] to collect information about the shared memory behavior in UPC benchmarks.

The PEM framework consists of four components: (i) an XML specification for events, (ii) a tool-set to generate stubs for both event generation and event consumption, (iii) an API that allows event selection and collection, and (iv) a runtime that implements the API. For this study we created a new XML specification for events related to allocating and accessing UPC shared variables.

We manually instrumented the UPC RTS using the event generation stubs created by the PEM tools. These stubs track allocation of shared objects and shared memory accesses. In each run we logged the following information for each shared-array-element access: the SVD entry of the shared array being accessed, the thread that owns the array element, the thread that is accessing the array element and the type of access (load or store). By recording the thread that owns the element, rather than the shared-memory domain of the element, we are able to determine how many of the shared accesses will be local in different machine configurations. The SVD entry for each shared array provides a unique key that is used to identify each shared array. Each shared-object allocation was also monitored to record the SVD entry for every shared variable when it is allocated. This monitoring allows us to manually associate shared accesses in a trace file (each trace contains the SVD entry of the shared array being accessed) with shared variables in the source code. This monitoring step could be automated if we modified the compiler to generate calls to the PEM tools to associate shared variables with SVD entries. This compiler modification has been left for future work.

Benchmarks were compiled with the UPC compiler and linked with the instrumented RTS library. Once the benchmarks were run, the PEM runtime was able to collect a trace of the events described above. We then implemented a PEM consumer to process and analyze these traces. This tool collected statistics about the shared array accesses for each shared array and each UPC thread in a given trace.

2.3 Terminology

In order to facilitate understanding the discussion in the following sections of the paper, we define the terms below.

- A *thread* T refers to a UPC-declared thread of execution.
- A *processor* P is a hardware context for executing threads. Multiple threads can be executed on one processor[3].
- A *node* is a collection of processors that access a shared and coherent section of the address space.
- A *thread group* is a collection of threads that execute in the same node (the software equivalent of a node).
- A *shared-memory domain* is the shared memory in a node that is common to a thread group.
- Each element of a shared array is a *shared array element*.
- A *shared array access* is a dynamic memory access to a shared array element.
- A thread T *owns* an element of a shared array if the location of the element is in the shared memory pertaining to T (*i.e.*, the element has affinity to T).

[3] Thus a processor may be a context in a hyper-threading processor, or it may be a core in a chip-multiprocessor architecture, or it may be a stand-alone processor.

– The *local shared array elements* for a thread T are the array elements that are located in the shared-memory domain of T. These elements may be owned by T or may be owned by other threads that are in the thread group of T.
– The *remote shared array elements* for a thread T are the array elements that are outside the shared-memory domain of T. These are elements that are owned by threads outside of the thread group of T.

The `shared` keyword is used in UPC to identify data that is to be shared among all threads. Every shared object has affinity with one, and only one, thread. The programmer can specify the *affinity* between shared objects and threads using the blocking factor. If a shared object O_s has affinity with a thread T then T *owns* O_s. In an ideal UPC program, the majority of shared data accesses are to shared data owned by the accessing thread. Such a data distribution reduces the amount of data movement between threads, thereby improving the performance of the program.

2.4 Overview of current UPC Benchmarks

From the NAS suite [3, 4], we selected the CG, MG and IS kernels. These benchmarks were developed by the UPC group at George Washington University based on the original MPI+FORTRAN/C implementation [17]. Each kernel comes with three versions containing different levels of user-optimized code. We used the O0 versions of the kernels because they contain the least amount of hand optimizations. Each kernel also comes with several class sizes that dictate the input size used by the benchmark. When possible, each benchmark was run with input classes S, A and B. The memory requirements for class S are the smallest and for class B are the largest that we could run. Not all the benchmarks could be run with class B.

CG is a kernel typical of unstructured grid computations. CG uses a conjugate-gradient method to approximate the smallest eigenvalue in a large, sparse matrix. The matrix is evenly divided between the processors.

MG uses a multigrid method to compute the solution of the 3D scalar Poisson equation. The partitioning is done by recursively halving the grid until all the processors are assigned. This benchmark must be run in K processors where K must be a power of 2. Communication occurs between iterations by exchanging the borders.

Integer Sort (IS) performs a parallel sort over small integers. Initially the integers are uniformly distributed.

A Sobel Edge Detection benchmark, written for this study, was also used. The Sobel operator is a discrete differentiation operator used in image processing. It computes an approximation of the gradient of the image intensity function. At each point in an image, the result of the Sobel operator is either the corresponding gradient vector or the norm of the vector [1].

The remaining UPC NAS Benchmarks have been optimized for access locality through the use of UPC block memory transfer methods (*e.g.*, `upc_memget`, `upc_memput`, `upc_memcpy`). These benchmark versions contain a relatively small number of accesses to shared variables and may not be representative for this study. We

expect to use them as a target that our compiler should strive to achieve by analyzing naively written UPC programs. [4]

3 Results and Discussion

There are four questions that we are interested in answering in this study: (i) What is the ratio of local to remote shared array accesses? (ii) Of the remote accesses, what is the subset of threads that own them? (iii) Are there regular patterns in accessing remote data? (iv) How does the blocking factor used to distribute the shared arrays impact the ratio of local to remote accesses?

For each of these questions, we will take one of the benchmarks described above and discuss what are the characteristics that make it display a particular behavior. Given that the set of benchmarks available in UPC is quite restricted, we hope that our discussion tying program features to performance characteristics will also serve as a best-practice foundation for UPC programmers.

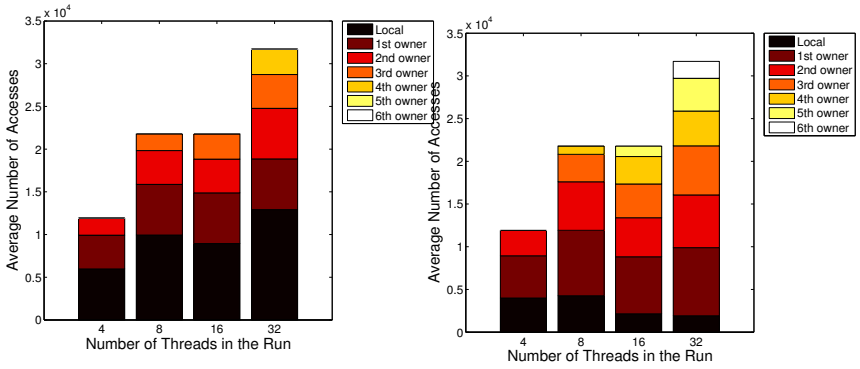
3.1 Local vs Remote Access Ratio

For the CG benchmark, more than 99.5% of the shared array accesses are to six shared arrays (independent of the number of threads used to run the benchmark): `send_start_g`, `exchange_len_g`, `reduce_threads_g`, `reduce_send_start_g`, `reduce_send_len_g`, and `reduce_recv_start_g`. The `send_start_g` and `exchange_len_g` are shared arrays with `THREADS` elements that are used in calls to `upc_memget` to move shared data between processors. They are created with the default (cyclic) blocking factor, where each processor is assigned one array element in a cyclic fashion. The `reduce_threads_g`, `reduce_send_start_g`, `reduce_send_len_g`, and `reduce_recv_start_g` are two-dimensional shared arrays of size `THREADS*NUM_PROC_COLS`, where `NUM_PROC_COLS` is based on the class size; the arrays use a blocking factor of `NUM_PROC_COLS`. These arrays are used in the conjugate-gradient computation.

Threads access only the elements they own in the `reduce_recv_start_g` and `reduce_threads_g` shared arrays. For the `reduce_send_len_g` and `exchange_len_g` shared arrays almost all accesses are to remote array elements. The local access ratios for `send_start_g` and `exchange_len_g` vary between threads. For example, for Class B run with 16 threads, threads 0, 5, 10 and 15 only access local array elements and the remaining threads access almost exclusively remote elements.

Figure 1(a) shows the distribution of array element accesses vs. ownership for accesses performed by the CG Benchmark running with Class B input. For thread i, we record all the threads that own elements accessed by i. We sort the threads in descending order of the frequency of accesses. The bars in the graph show, for each run, how many elements were accessed in one of the other threads, averaged over all threads. For example, when run with 32 threads, about 41% of accesses are local, 19% are to

[4] The LU benchmark does not currently verify when compiled with our compiler and thus was not included in the study.



(a) Blocking Factor = NUM_PROC_COLS

(b) Blocking Factor = 1

Fig. 1. Threads involved in Remote Accesses for blocking factor of NUM_PROC_COLS and for a blocking factor of 1 in the CG Class B benchmark.

a remote thread (first owner), 19% to a second owner, 12% to a third owner and about 9% to a fourth. This ownership distribution indicates that most of the remote accesses are confined to a small number of remote threads: even when run with 32 threads the majority of remote accesses are to at most 4 unique threads. Almost all benchmarks that we studied exhibit this type of pattern for up to 128 threads. Of the benchmarks we analyzed, IS is the only one that does not exhibit similar behavior. In IS, approximately 40% of remote accesses are to a large number of threads.

Table 1 shows the number of local accesses as a percentage of the total number of shared accesses for each benchmark run with the number of threads and the number of threads per group (TpG) specified. In most of these benchmarks, a large number of accesses are local (more than 40%) even when there is a single thread in each thread group. From these accesses, the ones in CG, MG and Sobel are mostly easily detected by the compiler. Therefore, they can be privatized to avoid the overhead of translation through the SVD.

The results in Table 1 indicate that as the benchmarks are run with more threads per group, the percentage of local shared accesses increases. These results highlight the benefit of running UPC programs on architectures that have multiple processors in each share-memory domain. For the CG and Sobel benchmarks, an overwhelming majority (more than 90%) of the accesses become local when run with a thread group consisting of 50% of the running threads. Even the IS benchmark, which exhibits irregular shared-access patterns improves significantly as the size of the thread groups is increased. In the case the compiler fails to identify and privatize these additional local accesses, the performance will still improve because it is not necessary to send messages between the accessing thread and the owning thread in order to exchange shared data.

Benchmark	UPC Threads	Percentage of Local Shared Accesses				
		1 TpG	2 TpG	4 TpG	8 TpG	16 TpG
CG Class B	4	50.2	83.4	-	-	-
	8	45.6	72.8	90.9	-	-
	16	41.1	68.3	86.4	90.9	-
	32	40.8	59.5	78.2	90.6	93.8
IS Class S	2	50.0	-	-	-	-
	4	25.1	50.0	-	-	-
	8	13.2	25.2	50.1	-	-
	16	7.6	13.7	25.7	50.5	-
	32	6.2	9.3	15.2	27.1	51.4
MG Class S	2	74.8	-	-	-	-
	4	62.2	74.8	-	-	-
	8	55.4	62.3	74.9	-	-
	16	52.3	56.0	62.3	74.9	-
	32	50.6	52.9	56.1	62.5	75.0
Sobel Easter (BF 1)	2	26.68	-	-	-	-
	4	23.3	60.0	-	-	-
	8	21.7	56.7	76.7	-	-
	16	20.8	55.0	73.3	85.0	-
	32	20.4	54.1	71.7	81.7	89.2
Sobel Easter (Max BF)	2	93.2	-	-	-	-
	4	89.7	93.2	-	-	-
	8	87.7	89.7	93.2	-	-
	16	86.2	87.7	89.7	93.2	-
	32	84.3	86.2	87.7	89.7	93.2

Table 1. Local accesses as a percentage of total shared accesses as a function of the number of UPC threads and the number of threads per group (TpG).

3.2 Remote Data Access Patterns

An important factor in deciding the mapping of threads to processors in a hybrid architecture is the access patterns to remote data. This pattern depends on the algorithm used for solving the problem. Here we present evidence that, for a number of algorithms used in scientific computations, regular access patterns indeed appear and these patterns are amenable to optimization.

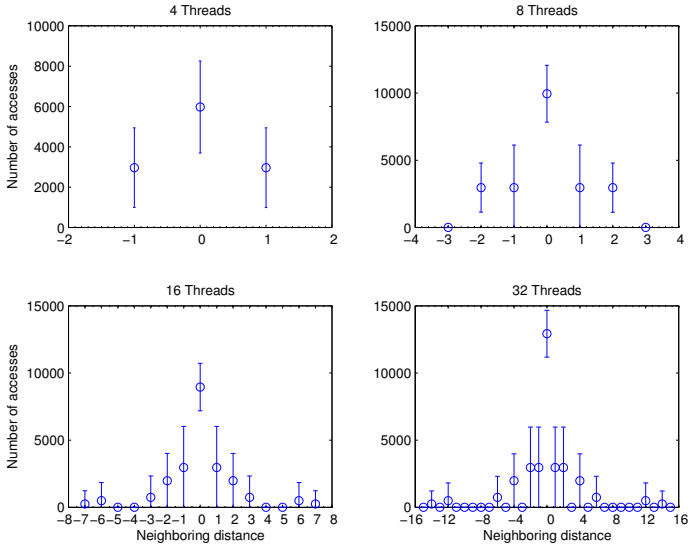


Fig. 2. Distance to remote accesses for CG Class B

In Figure 2 we capture the actual pattern of data exchange between threads for the CG Benchmark running the class B input. We assume the threads are cyclically distributed and we compute the *distance* between two threads as the number of threads separating them in a ring distribution (thread 0 comes after thread N-1). A distance of zero represents accesses to local data. In these error-bar plots the circles are the average number of accesses to a remote thread. The error bars are the standard deviations. For the 32-way CG, we know, from Figure 1(a), that most remote accesses occur to four other threads. In this figure, we observe that those threads are actually the immediate neighbors, that is, the threads at distances -2, -1, 1, and 2. Most of the other benchmarks show a similar behavior, except for IS, where the remote accesses are relatively uniformly spread throughout all threads. This pattern of communicating with a small number of threads increases the opportunities to, with a good thread/processor mapping, privatize local shared data accesses in hybrid architectures.

The different access distribution in IS (integer sorting) occurs because each thread owns a set of buckets and a random set of keys that need to be sorted. The behavior of the IS benchmark class S with 2 to 128 threads is shown in Figure 3.

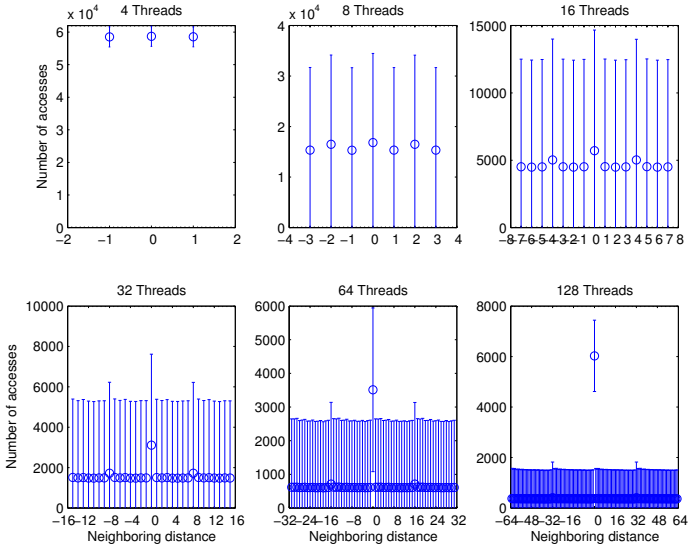


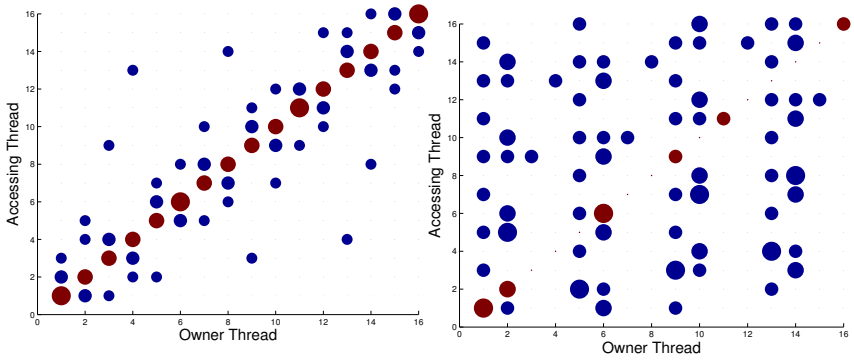
Fig. 3. Distance to remote accesses for IS Class S

The scatter plot in Figure 4(a) displays the distribution of local and remote accesses performed by each thread in the CG benchmark running with 16 threads and with class B input. The size of each point is proportional to the number of accesses performed by the accessing thread to shared array elements that are mapped to the owner thread. The colors highlight the local accesses.

Figure 4(a) shows that when a blocking factor of NUM_PROC_COLS is used the majority of shared memory accesses are clustered along the diagonal. Every access on the diagonal is a local access (accessing thread equals the owning thread) while accesses near the diagonal indicate the accessing and owning threads are in close proximity to each other (in terms of thread distance). This observation provides strong support for running the benchmark on a hybrid machine where threads are mapped to thread groups based on their distance from each other. For example, on a hybrid machine with thread groups of size four, threads 0 through 3 are mapped to one node, threads 4 through 7 are mapped to a second node, *etc.*,.

3.3 Effects of Blocking Factor

To illustrate the effect the blocking factor, the CG benchmark was modified to create the `reduce_threads_g`, `reduce_send_start_g`, `reduce_send_len_g`, and `reduce_recv_start_g` arrays with a blocking factor of 1. Figure 1(b) shows the number of unique threads involved in remote accesses while the scatter plot in Figure 4(b) shows the distribution of local and remote accesses. These figures emphasize the importance of using an adequate blocking factor to increase the number of shared accesses that are local and thus candidates for privatization. When compared with the

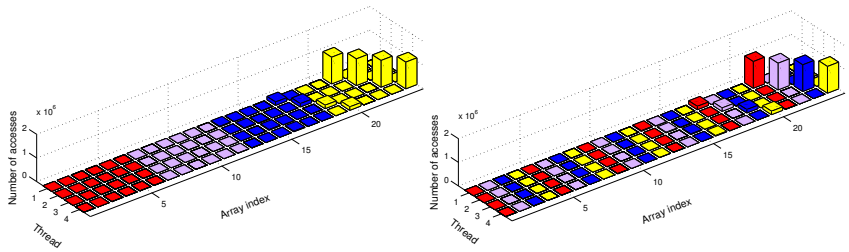


(a) Blocking Factor = NUM_PROC_COLS

(b) Blocking Factor = 1

Fig. 4. Distribution of shared accesses for blocking factor of NUM_PROC_COLS and for a blocking factor of 1 in the CG Class B benchmark running with 16 threads. The darker markers on the diagonal are local accesses, while the lighter colored markers are remote accesses. The size of the marker denotes the number of accesses, the larger, the more accesses.

plot in Figure 4(a), we see the selection of blocking factor is even more important for hybrid architectures.



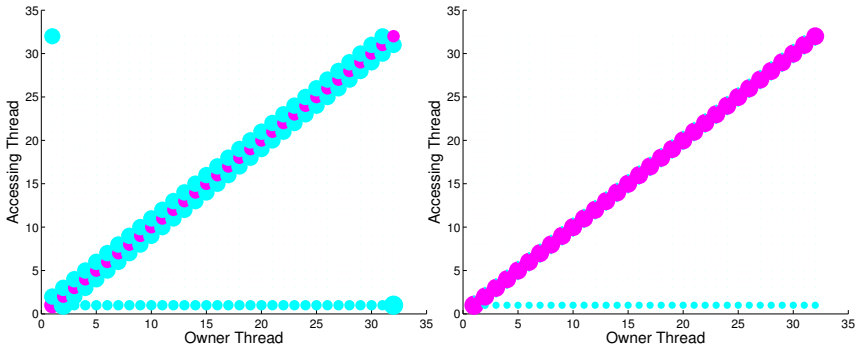
(a) Original Blocking Factor

(b) Blocking Factor = 1

Fig. 5. MG class S array access index frequency using original blocking factor. Color denotes ownership.

In the NAS MG Benchmark, we observed a high percentage of remote accesses for the two shared arrays `sh_u` and `sh_r`. These two shared arrays contain the original and

residual matrices used in the multigrid computation. They are relatively small arrays (for class S their size is 6*THREADS) and they are distributed using a blocking factor of 6. Figure 5(a) shows the index frequency histogram for the sh_u array when run with input class S on 4 threads. The height of the bars indicate the number of accesses to a specific index. The colors denote the ownership of the shared data being accessed. From this histogram we see that the majority of accesses are to indices 20, 21, 22 and 23. However, because a blocking factor of 6 was used to distribute the array, all of these indices map to thread 3. When run with the original blocking factor there were approximately 12.5% local accesses to sh_u. By manually modifying the source code to use the default blocking factor of 1, the number of local accesses increased to 99.4%. The index frequency histogram using a blocking factor of 1 is shown in Figure 5(b).



(a) Blocking Factor = 1 (b) Blocking Factor = MAX_BLOCK

Fig. 6. Distribution of shared accesses for blocking factor of 1 and blocking factor of MAX_BLOCK = (ROWS*COLUMNS)/THREADS for the Sobel benchmark running with 32 threads.

Figure 6 illustrates the effect of different blocking factors on the Sobel benchmark. The Sobel benchmark performs a stencil computation where the eight neighbors of an array element are used to compute a new value. Thus choosing the largest possible blocking factor, such that the eight neighbours of a give array element are local (for most array elements) proves to be the best strategy.

4 Related Work

This is the first performance study to provide an empirical measurement of the distribution of thread ownership of the accesses performed by each thread in UPC benchmarks. This data allows the community to both identify the opportunities for optimization of data accesses in UPC and to estimate the potential gains that such optimizations may yield.

Several research groups have investigated the performance of UPC and compared with other languages. El-Ghazawi and Cantonnet found that both the Compaq UPC 1.7 Compiler on the Compaq AlphaServer SC produced code that was slower than, but competitive with, the code produced for MPI versions of the NAS parallel benchmarks [17]. In their study the UPC codes were modified by hand to convert all local shared accesses into private accesses.

The performance study by Cantonnet *et al.* provides strong support to improving the code generation for local shared accesses in UPC[8]. Not only did they measure the high overhead of such accesses in current compilers, they also demonstrated, through hand-optimization, that significant performance gains can be obtained by privatizing such accesses. By manually privatizing local shared accesses and prefetching remote data in the Sobel Edge Detection benchmark they were able to obtain nine times speedup for a 2048×2048 image and results showing very-high parallel efficiency with speedup almost linear on the number of processors. In cluster architectures formed by many multiprocessor nodes the potential for improvement in performance may be even higher than these experiments indicate. Similarly Chen *et al.* found that if local shared accesses were privatized in the Berkeley UPC compiler, a simple vector addition application would see an order of magnitude speedup [11].

Berlin *et al.* compared the time required for a private local access and for a shared local access [6]. The smallest difference between these two accesses was in the SGI Origin 2000 (the private access was 7.4 times faster). In a 64-node Compaq AlphaServer cluster with four ES-40 processors per node with an earlier version of the Compaq UPC compiler, they found that a private access was 580 times faster than a local shared access in the same processor, and was 7200 times faster than a shared access to an address mapped to another processor in the same node. Later versions of that compiler have reduced this overhead, but these staggering numbers speak to the need to improve the identification and privatization of local-shared accesses.

In a comparative study between UPC and Co-array Fortran, Coarfa *et al.* found that in both languages, bulk communication is essential to deliver significant speedup for the NAS benchmarks [12]. They also point out that references to the local part of shared array through shared pointers is a source of inefficiency in UPC. They suggest that the way to work around this inefficiency is for UPC programmers to use private C pointers to access the local part of shared objects. We propose a more elegant two-pronged solution: (1) an optimizing UPC compiler may modify the blocking factor to improve the number of local accesses for a given machine configuration; and (2) the compiler should automatically convert shared accesses to the local part of a shared array into private local accesses.

The analysis required for the privatization of local shared accesses finds parallel in the analysis of array accesses in loop nests in the modified version of Parascope by Dwarkadas *et al.* [16]. Their goal is to inform the runtime system that it does not need to detect accesses to shared data. Their compiler-time analysis allows the runtime system to prepare for the shared accesses ahead of time. In UPC the analysis will be able to simply replace the shared access with a simple pointer-based access.

Zhang and Seidel developed the UPC STREAM benchmark [23]. Their experimental study also found the overhead of accessing local sections of a shared array through

shared accesses to be significant. They also report on an empirical comparison between an implementation of UPC over MPI and Pthreads from Michigan Technological University, the first commercial UPC compiler from Hewlett-Packard, and the Berkeley UPC compiler.

Zhang and Seidel propose a model to predict the runtime performance of UPC programs [24]. Their technique uses dependence analysis to identify four types of shared memory access patterns and predict the frequency of each pattern. Microbenchmarks are used to determine the cost of each pattern on various architectures. Their results demonstrate their model is able to predict execution times within 15% of actual running times for three application benchmarks.

The issue of identifying shared accesses that are local arises in UPC because of a language design decision that makes the physical location of the memory referenced transparent to the programmer. Other languages, such as Co-Array Fortran [20] expose the distinction between local and remote accesses at the language level and thus they compilers do not have to deal with the privatization of local shared accesses.

Barton *et al.* describe a highly scalable runtime system based on the design of a shared variable directory [5]. They also describe the optimization of the `upc_forall` loop, local memory accesses, and remote update operations implemented in the IBM XL UPC Compiler [14].

Intel has recently announced Cluster OpenMP that support OpenMP programs running on a cluster of workstations [18]. A licensed version of TreadMarks [2] is used for the runtime system to manage the movement of shared data between nodes. The OpenMP specification has been extended to include the *sharable* directive, used to identify variables shared among threads. A sharable equivalent to malloc has also been added to support dynamic shared data.

A mixed-mode programming model for hybrid architectures has been explored by several groups. In this mixed-mode model, MPI is used to communicate between nodes while OpenMP is used to parallelize work within a node. Smith and Bull conclude that this mixed-mode programming model is well suited for some applications but warn it is not the best solution for all parallel programming problems [21]. A programmer should understand the nature of the application (load balancing characteristics, parallel granularity, memory limitations from data replication and general MPI performance) before attempting to use the mixed-mode model.

5 Conclusions

We started this detailed study of data access patterns in UPC from the premise that understanding these patterns will allow us to estimate the potential of several compiler optimizations. Indeed, we find that the number of local accesses that are identifiable by the compiler is quite high in the set of benchmarks that we studied. Privatizing these accesses automatically will remove a significant source of overhead while keeping the code portable and simple to understand.

In addition, we observed that, contrary to the intuition that the largest blocking factor is always better for improving locality, there are cases in which a blocking factor

selected based on the access pattern provides more benefit. We are working on a solution to the problem of finding the best blocking factor for an application.

And finally, we have shown that even considering a naive mapping of threads to processors in a hybrid architecture, there is tremendous potential to increase the performance of applications because of better data locality. We are confident that through a combination of compiler and runtime optimization the performance of PGAS languages such as UPC can be on-par with traditional high performance MPI+OpenMP codes. At the same time, PGAS programming models are a more elegant solution to the problem of programming hybrid architectures when compared with mixed programming models, such as combinations of MPI with OpenMP.

References

1. Sobel - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Sobel.
2. Christina Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.
3. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R.Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.
4. David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995.
5. Christopher Barton, Calin Cascaval, George Almasi, Yili Zhang, Montse Farreras, Siddhartha Chatterjee, and José Nelson Amaral. Shared memory programming for large scale machines. In *Programming Language Design and Implementation (PLDI)*, pages 108–117, June 2006.
6. Konstantin Berlin, Jun Huan, Mary Jacob, Garima Kochbar, Jan Prins, Bill Pugh, P. Sadayappan, Jaime Spacco, and Chau-Wen Tseng. Evaluating the impact of programming language features on the performance of parallel applications on cluster architectures. In *The 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003)*, Texas, USA, October 2003.
7. Dan Bonachea. GASNet specification, v1.1. Technical Report CSD-02-1207, U.C. Berkeley, November 2002.
8. François Cantonnet, Yiyi Yao, Smita Annareddy, Ahmed S. Mohamed, and Tarek A. El-Ghazawi. Performance monitoring and evaluation of a UPC implementation on a NUMA architecture. In *International Parallel and Distributed Processing Symposium (IPDPS)*, page 274, Nice, France, April 2003.
9. Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks. In *ACM/IEEE Supercomputing*, November 2000.
10. Calin Cascaval, Evelyn Duesterwald, Peter F. Sweeney, and Robert W. Wisniewski. Performance and environment monitoring for continuous program optimization. *IBM Journal of Research and Development*, 50(2/3), March 2006.
11. Wei-Yu Chen, Dan Bonachea, Jason Duell, Parry Husbands, Costin Iancu, and Katherine Yelick. A performance analysis of the Berkeley UPC compiler. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS'03)*, June 2003.

12. Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: Co-Array Fortran and Unified Parallel C. In *Symposium on Principles and Practice of Parallel Programming(PPoPP)*, pages 36–47, Chicago, IL, June 2005.
13. UPC Consortium. *UPC Language Specification, V1.2*, May 2005.
14. IBM Corporation. IBM XL UPC compilers. http://www.alphaworks.ibm.com/tech/upccompiler, November 2005.
15. J.J. Costa, Tony Cortes, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. Running OpenMP applications efficiently on an everything-shared SDSM. *Journal of Parallel and Distributed Computing*, 66:647 – 658, September 2005.
16. Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Architectural Support for Programming Languages and Operating Systems*, pages 186–197, Cambridge, MA, 1996.
17. Tarek El-Ghazawi and François Cantonnet. UPC performance and potential: a NPB experimental study. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing (SC'02)*, pages 1–26, Baltimore, Maryland, USA, November 2002.
18. Jay P. Hoeflinger. Extending OpenMP to clusters. 2006. http://www.intel.com/cd/software/products/asmo-na/eng/compilers/285865.htm.
19. Pete Keleher. *Distributed Shared Memory Using Lazy Release Consistency*. PhD thesis, Rice University, December 1994.
20. Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *ACM SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.
21. Lorna Smith and Mark Bull. Development of mixed mode mpi / openmp applications. *Scientific Programming*, 9(2-3/2001):83–98. Presented at Workshop on OpenMP Applications and Tools (WOMPAT 2000), San Diego, Calif., July 6-7, 2000.
22. Robert W. Wisniewski, Peter F. Sweeney, Kertik Sudeep, Matthias Hauswirth, Evelyn Duesterwald, Călin Caşcaval, and Reza Azimi. Performance and environment monitoring for whole-system characterization and optimization. In *PAC2 - Power Performance equals Architecture x Circuits x Compilers*, pages 15–24, Yorktown Heights, NY, October 6-8 2004.
23. Zhang Zhang and Steven Seidel. Benchmark measurements of current UPC platforms. In *International Parallel and Distributed Processing Symposium (IPDPS)*, Denver, CO, April 2005.
24. Zhang Zhang and Steven Seidel. A performance model for fine-grain accesses in UPC. In *International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, April 2006.