Sparse Matrix Multiplication Using UPC

Hoda El-Sayed and Eric Wright
Department of Computer Science
Bowie State University

***Abstract*—Partitioned global address space (PGAS) languages, such as Unified Parallel C (UPC) have the promise of being productive. Due to the shared address space view that they provide, they make distributing data and operating on ghost zones relatively easy. Meanwhile, they provide thread-data affinity that can enable locality exploitation. In this paper, we are considering sparse matrix multiplication which is an important operation for many scientific and engineering applications. Recently, several different high-performance algorithms and libraries have been developed for that operation. However, in this work, we were able to take advantage of one of the advanced features provided by UPC, which is the fact that it is a globally addressable memory model. Due to that feature, using UPC in this operation would enable threads to read or write data from any other thread's memory directly without any inter-process communication as the case with MPI. Our goal is to evaluate the performance of both parallel programming models based on experimental evaluation of sparse matrix multiplication. The comparative analysis will consider conceptual complexity and execution time. It will be shown that UPC which supports distributed shared memory model has a great productivity advantage over message passing when sparse matrix multiplication problems are considered.**

***Index Terms*—PGAS, UPC, MPI, and Sparse matrix**

I. INTRODUCTION

The capability of supercomputers/high performance computers has grown significantly over the past decade. During this period, the development of complex parallel programming paradigms required for evolving parallel computing architectures has made it increasingly difficult to develop high performance computing applications. Most high performance computing systems today are distributed memory systems, where each CPU has its own local memory. It takes much longer for a CPU to get data from another CPU's than from its own. One of the major challenges of efficient parallel programming is dealing with the relatively slow transfer of data from other CPU's memory. The performance of parallel programs is limited by the speed of the communication between the CPUs[4]. The two most popular parallel programming paradigms available are message-passing and shared memory. Both models usually require domain patitioning and load. Message passing requires distributing data structures across the processes and explicitly handling interprocess communications. While in shared memory, manipulating the data may require synchronization in addition to the lack of of data locality exploitation. Performance decreases due to the communication overhead which could be substantially large especially in small messages [8] and effects of synchronization. An emerging programming model is the global address space. UPC is a new parallel programming language that considers a partitioned global address space (PGAS) and provides ease of programming shared memory paradigms and also enables the exploitation of data locality. Due to the shared address space view that they provide, they make distributing data and operating on ghost zones relatively easy. Meanwhile, they provide thread-data affinity that can enable locality exploitation. The success of any programming language depends on many factors. Some of the important factors are the ease of use, efficient execution, and foundation on a good programming model.

In our work, we are considering sparse matrix multiplication which is an important operation for many scientific and engineering applications. Recently, several different high-performance algorithms and libraries have been developed for that operation. However, we were able to take advantage of one of the advanced features provided by UPC, which is the fact that it is a globally addressable memory model. Due to that feature, using UPC in this operation would enable threads to read or write data from any other thread's memory directly without any inter-process communication as the case with MPI.

Our goal is to evaluate the performance of both parallel programming models based on experimental evaluation of sparse matrix multiplication. The comparative analysis will consider conceptual complexity and execution time. It will be shown that UPC which supports distributed shared memory model has a great productivity advantage over message passing when sparse matrix multiplication problems are considered.

This paper is organized as follows: Section 2 presents an overview of UPC and the sparse matrix multiplication implementation using UPC, while section 3 shows the sparse matrix multiplication implementation using MPI. Section 4 presents the experimental performance and analysis measurements, followed by the conclusions in section 5.

II. UNIFIED PARALLEL C (UPC)

Unified Parallel C (UPC) is a new parallel programming language that extends ISO C with support for parallel processing. UPC is based on partitioned global address space (PGAS) and provides ease of programming shared memory paradigms and also enables the exploitation of data locality.

Due to the shared address space view that they provide, they make distributing data and operating on ghost zones relatively easy. Meanwhile, they provide thread-data affinity that can enable locality exploitation. In other words, it enables processors to read or write data from any other processor's memory.

UPC keeps the powerful concepts and features of C and adds parallelism; global memory access with an understanding of what is remote and what is local; and the ability to read and write remote memory with simple statements. The simplicity, usability and performance of UPC have attracted interest from high performance computing users and vendors. This interest resulted in vendors developing and commercializing UPC compilers [7]. Many parallel applications could scale more efficiently through the use of the high performance communication enabled by UPC. Moreover, the development cycle for many parallel applications could be dramatically reduced as a result of the clear, simple syntax that is inherent to UPC.

A. UPC Programming Model

UPC is a distributed shared memory programming model. The memory under UPC is logically divided into a shared memory space and a private memory space. Each thread has a private memory space that can be accessed only by that thread. The shared memory space is logically partitioned into portions each of which has a logical association, or affinity to a given thread. The entire shared memory space, regardless of affinity, can be accessed by all threads [7]. Figure 1 shows the memory layout for UPC. The number of threads is given by the special constant THREADS[5], and each thread is identified by the special constant MYTHREAD [2].

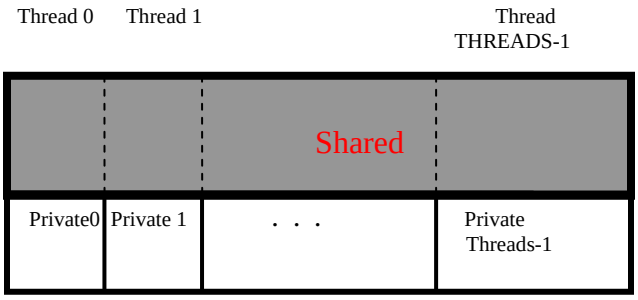


Figure1.
Distributed Shared Memory

A UPC shared pointer can reference all locations in the shared space; while a private pointer may reference only addresses in its private space or in its local portion of the shared space. Data in the local portion may only be accessed by the thread to which they have affinity, while data in the shared portion are accessible by all threads. Threads access shared memory addresses concurrently through standard read and write instructions, rather than through explicit message passing [10].

B . UPC Implementation

In this application, two sparse matrices A and B are to be multiplied together and the result is stored in matrix C. Matrix A is distributed by rows among the threads, while matrix B is distributed by columns among the threads. Since the UPC memory model provides a distributed shared view to the programmer, any element in any of the matrices could be accessed by any thread. The matrices could be distributed in such a way such that the first row in matrix A would be local to thread 0, and the second column in matrix B would be local to thread 1. To obtain a column in matrix C, we would multiply the row elements which are local to thread 0 by the column elements that are local to thread 1. However, when doing so using UPC, we would first read the row elements in matrix A. If any were zero, then the multiplication would be skipped. In other words, multiplication would take place only for nonnegative zero elements. In that sense, thread 0 would access the column elements local to thread 1 so easily as they are as well global to all other threads. That could only take place when we are using UPC since it is global addressable memory model. Therefore, by using UPC in this operation, threads would be able to read or write data from any other thread's memory directly without any inter-process communication as the case with MPI. As a result, a great deal of computations could be saved since we are using sparse matrices where most of the elements in the matrices are zeros.

In MPI, P1 would have to send the entire 2nd column to P0 for the multiplication to take place. That would still have to happen even if the elements of the 1st row in array A were zeros, (still P1 would send the column)

In UPC, only the nonzero elements in the rows of array A are the ones to be multiplied by the columns in array B. As the elements in array B are seen by all the threads because they are stored in shared memory.

C. Message Passing Interface (MPI)

The MPI is a library that considers message-passing the parallel programming paradigm. Calls to invoke the library routines are bulky and non-intuituve. Every process has its own address space, so processes exchange data between each other through explicit message passing. Each thread basically executes on its own CPU. The threads are assigned a unique number called rank, ranging from 0 to P-1, where P is the number of threads running. During communication between the threads, the threads specify the rank in their sending and receiving messages. Processes communicate and synchronize with each other using point-point or collective communication. The point-to-point operations allow two processes to exchange

data. The core functions are MPI_Send( ) and MPI_Receive( ). These functions require a pointer to a buffer that will be used for sending or receiving data. MPI_Send( ) requires an explicit source or MPI_ANY_SOURCE( ). MPI allows blocking and nonblocking communication. In collective communication, all processes or a subset of processes are involved in a particular operation. For example, MPI_Barrier( ) is used to synchronize all processes at a particular point in the code. The MPI_Bcast( ) function allows one process to broadcast data to all other participating processes[1]. The two-sided communication between processes results in substantial overhead in interprocessor communications, especially in the case of small messages. The complexity of communication involved on distributed memory computer systems would require more complex code to achieve satisfactory performance [5].

III. MPI IMPLEMENTATION

MPI is based on a master-slave process communication. The master process will distribute matrix A and matrix B among the different processes which act as the slaves. Every process will have its own chunk of rows from matrix A and columns from matrix B that are only accessible by the process. Matrix C will be computed from the multiplication of rows in matrix A by the columns in matrix B. So if for example we multiply the row elements of process 0 by the column elements of process 1, process 1 would have to send the entire row elements to process 0, and this has to happen even if most of the row elements in process 0 were zeros. However, in UPC, multiplication would be skipped if the row elements were zeros. In MPI inter-process communication would have to occur and that results in a large number of computations even if most of the elements were zeros. On the other hand, those computations were skipped when the row elements were zeros when UPC was used. That certainly affects the performance and causes extreme communication delay.

IV. PERFORMANCE EVALUATION

In this paper, we are considering sparse matrix multiplication which is an important operation for many scientific and engineering applications. Recently, several different high-performance algorithms and libraries have been developed for that operation. However, in this work, we were able to take advantage of one of the advanced features provided by UPC, which is the fact that it is a globally addressable memory model. Due to that feature, using UPC in this operation would enable threads to read or write data from any other thread's memory directly without any inter-process communication as the case with MPI. Our goal is to evaluate the performance of both parallel programming models based on experimental evaluation of sparse matrix multiplication. Performance is assessed based on wall clock time measurements on an SGI Origin distributed shared memory system. The comparative analysis will consider conceptual complexity and execution time. The execution time was used to evaluate the speedup using UPC and MPI programming models. Different matrices sizes were used on different number of processes. From the execution times, we were able to observe that UPC took much less execution time than MPI, and that is due to the fact that in UPC we were able to skip the computations with zero elements. However, when MPI was used, communication was involved even with the zero elements, still the entire column had to be sent to the other processor to perform the multiplication.

Figure 4 shows the speedup using MPI, while figure 5 shows the speedup using UPC programming model. The figures show that both models scale. MPI seems to scale better than UPC based on the figure, however, examining the execution time in figures 1 and 2, it shows that the UPC execution time was significantly smaller than the MPI. MPI was able to hide more interprocess communication overhead and artificially appear as if it were performing better. However, UPC was able to minimize data transfer which has resulted in its faster execution time. From the results, we were able to conclude that UPC which supports distributed shared memory model has better performance over message passing when sparse matrix multiplication problems are considered.

Figure 2: Execution Time Using MPI

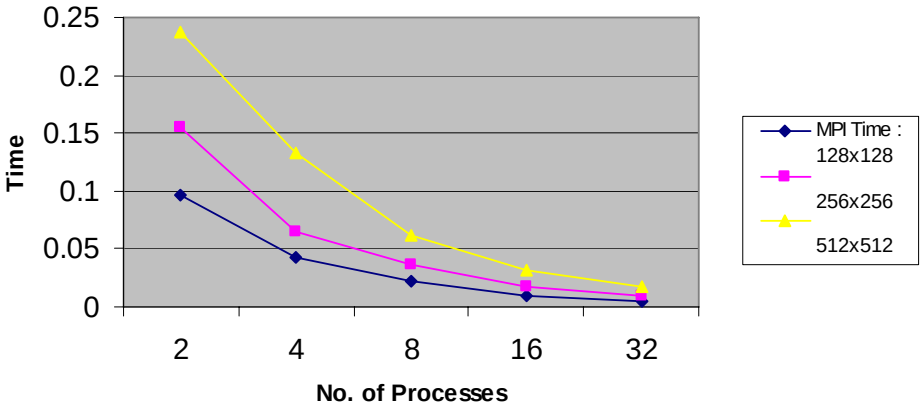


Figure 3: Execution Time Using UPC

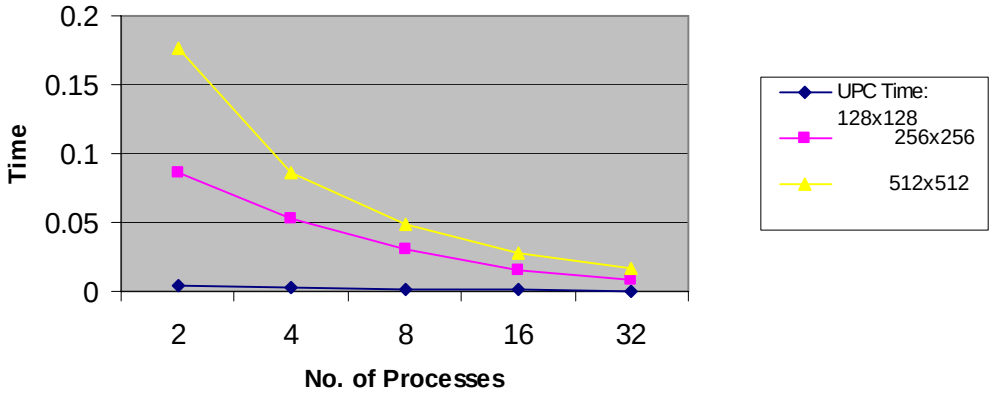


Figure 4: SpeedUp Using MPI

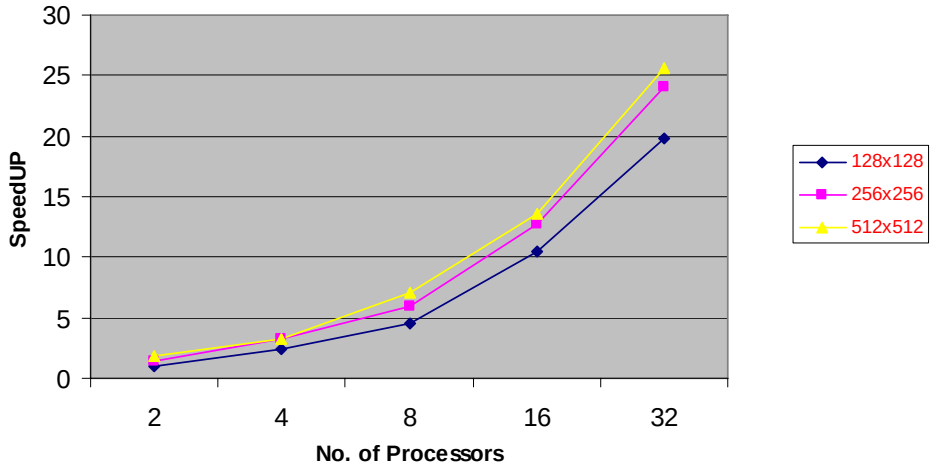
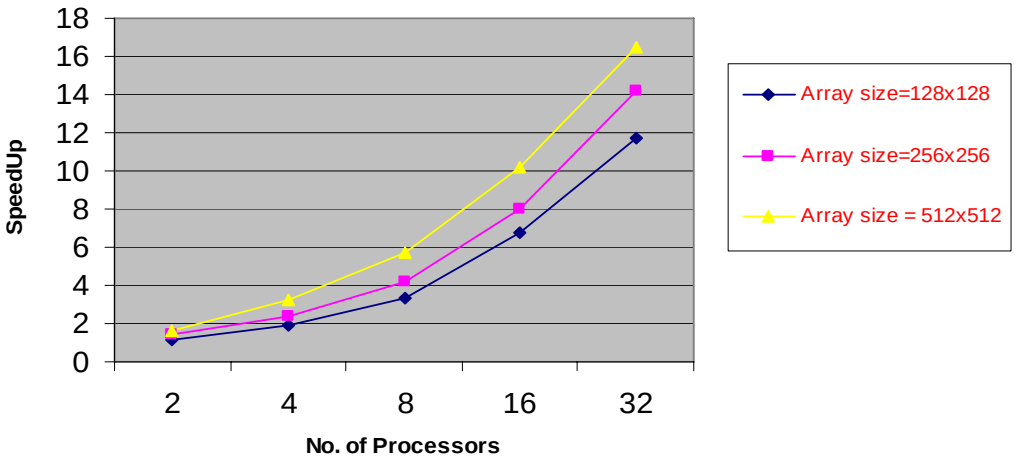


Figure 5: UPC SpeedUp



V. CONCLUSION

Partitioned global address space (PGAS) languages, such as Unified Parallel C (UPC) have the promise of being productive. Due to the shared address space view that they provide, they make distributing data and operating on ghost zones relatively easy. Meanwhile, they provide thread-data affinity that can enable locality exploitation. In this paper, we are considering sparse matrix multiplication which is an important operation for many scientific and engineering applications. We were able to take advantage of that feature provided by UPC, in the sparse matrix multiplication operation as it would enable threads to read or write data from any other thread's memory directly without any inter-process communication as the case with MPI. By that all the zero elements were skipped from the multiplication operation when using UPC, however, in MPI, the entire column still had to be sent to the other process for multiplication even though most of the elements were zeros. That obviously means more communication and in turn lowers the performance. MPI seems to scale better than UPC based on the figures, however, examining the execution time, it turns out that the UPC execution time was significantly smaller than the MPI program due to its longer execution time, MPI was able to hide more interprocess communication overhead and artificially appear as if it were performing better. However, UPC was able to minimize data transfer which has resulted in its faster execution time. From the results, we were able to conclude that UPC which supports distributed shared memory model has better performance over message passing when sparse matrix multiplication problems are considered. It has shown based on the performance measurements that UPC provides an effective alternative for programming parallel sparse matrix multiplication.

REFRENCES

[1] S. Caglar, G. Benson, Q. Huang, and C. Chu. *USFMPI: A Multi-threaded Implementation of MPI for Linux Clusters*. <www.cs.usfca.edu/~qhuang/papers/usfmpi/pdf, Jan. 2005.

[2] F. Cantonnet, Y. Yao, S. Annareddy, A. Mohamed, and T. El-Ghazawi. "*Performance Monitoring and Evaluation of a UPC Implementation on a NUMA Architecture*", IPDPS 2003.

[3] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi, "*Productivity Analysis of the UPC Language*", IPDPS 2004 PMEO Workshop.

[4] J. Dawson, "*Co-Array Fortran for Productivity and Performance*", Army HPC Research Center Bulletin, Vol. 14 No. 4, 2004.

[5] T. El-Ghazawi, W. Carlson, and J. Draper, "*UPC Language Specifications*", V1.1.1 (http://upc.gwu.edu), October 2003.

[6] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. "*UPC Distributed Shared Memory Programming*", John Wiley & Sons Inc. Publication, 2005.

[7] T. El-Ghazawi and S. Chauvin, " *UPC Benchmarking Issues*", 30th Annual Conference IEEE International Conference on Parallel Processing, 2001 (ICPP01), pp. 365-372.

[8] T. El-Ghazawi and F. Cantonnet, "*UPC Performance and Potential: A NPB Experimental Study*", SuperComputing 2002.

[9] W. Kuchera and C. Wallace. "*The UPC Memory Model: Problems and Prospects*", PDPS 2004.