Towards a Complexity Model for Design and Analysis of PGAS-Based Algorithms

Mohamed Bakhouya[1], Jaafar Gaber[2], and Tarek El-Ghazawi[1]

[1] Department of Electrical and Computer Engineering
High Performance Computing Laboratory
The George Washington University
{bakhouya,tarek}@gwu.edu
[2] Universite de Technologies de Belfort-Montbeliard
gaber@utbm.fr

Abstract. Many new Partitioned Global Address Space (PGAS) programming languages have recently emerged and are becoming ubiquitously available on nearly all modern parallel architectures. PGAS programming languages provide ease-of-use through a global shared address space while emphasizing performance by providing locality awareness and a partition of the address space. Examples of PGAS languages include the Unified Parallel C (UPC), Co-array Fortran, and Titanium languages. Therefore, the interest in complexity design and analysis of PGAS algorithms is growing and a complexity model to capture implicit communication and fine-grain programming style is required. In this paper, a complexity model is developed to characterize the performance of algorithms based on the PGAS programming model. The experimental results shed further light on the impact of data distributions on locality and performance and confirm the accuracy of the complexity model as a useful tool for the design and analysis of PGAS-based algorithms.

1 Introduction

PGAS implicit communication and fine-grain programming style make application performance modelling a challenging task [2], [5]. As stated in [2] and [5], most of the efforts have gone into PGAS languages design, implementation and optimization, but little work has been done for the design and analysis of PGAS-based algorithms. Therefore, a complexity model for PGAS algorithms is an interesting area for further research and design, and this field, to the best of our knowledge, had not yet been adequately defined and addressed in the literature. The challenge is to design, analyze, and evaluate PGAS parallel algorithms before compiling and running them on the target platform.

Recall that in sequential programming, assuming that sufficient memory is available, execution time of a given algorithm is proportional to the work performed (i.e., number of operations). This relationship between time and work makes the performance analysis and comparison very simple. However, in parallel performance models proposed in literature, other platform-dependent parameters, such as communication overhead, are used in developing parallel programs

R. Perrott et al. (Eds.): HPCC 2007, LNCS 4782, pp. 672–682, 2007.

[8]. By changing these parameters, developers can predict the performance of the programs on different parallel machines. However, these parameters are not program-dependent and considering them in the design phase can complicate the analysis process. In other words, having platform-dependent parameters in the model makes it quite difficult to obtain a concise analysis of algorithms. To meet this requirement, complexity models are useful for programmers to design, analyze, and optimize their algorithms in order to get better performance. They also provide a general guideline for programmers to choose the better algorithm for a given application.

This paper addresses a complexity model for the analysis of the intrinsic efficiency of the PGAS algorithms with disregard to machine-dependent factors. The aim of this model is to help PGAS parallel programmers understand the behavior of their algorithms and programs, making informed choices among them, and discovering undesirable behaviors leading to poor performance.

The rest of the paper is organized as follows. In section 2, we present related work. Section 3 presents an overview of PGAS programming model by focusing on one language implementing it, UPC. In section 4, the complexity model for PGAS-based algorithms is presented. Experimental results are given in section 5. Conclusion and future work are presented in section 6.

2 Related Work

There has been a great deal of interest in the development of performance models, also called abstract parallel machine models, for parallel computations [12]. The most popular is the Parallel Random Access Machine (PRAM) model, which is used for both shared memory and network-based systems [10]. In shared memory systems, processors execute in concurrently and communicate by reading and writing locations in shared memory spaces. In network-based systems, processors coordinate and communicate by sending messages to their neighbors in the underlying network (e.g., array, hypercube, etc) [6], [10]. While the PRAM model does not consider the communication cost [10], it is considered by many studies to be high level and unable to accurately model parallel machines. New alternatives such as Bulk Synchronous Parallel (BSP) [9], LogP and its variants [4],[8], and Queuing Shared Memory (QSM) [10] have been proposed to capture the communication parameters. These models can be grouped into two classes: shared memory-based models and message passing-based models. LogP and its variants, and BSP are message passing-based models that directly abstract distributed memory and account for bandwidth limitations. Queuing Shared Memory (QSM) is a shared memory-based model [10], [11]. Despite its simplicity it uses only machine-dependent parameters, and in addition the shared memory is global and not partitioned, i.e. it provides no locality awareness.

Recently, approaches to predict the performance of UPC programs and compiler have been proposed in [2] and [5]. In [5], authors include machine-dependent parameters in their model to predict UPC program performance. The model proposed in [2], offers a more convenient abstraction for algorithms design and

development. More precisely, the main objective of this modelling approach is to help PGAS designers to select the most suitable algorithm regardless of the implementation or the target machine. The model presented in this paper extends the modelling approach proposed in [2] by deriving analytical expressions to measure the complexity of PGAS algorithms. As stated in [2], considering explicitly platform parameters, such as the latency, the bandwidth, and the overhead, during the design process could lead to platform-dependent algorithms. Our primary concern in this paper is a complexity model for the design and analysis of PGAS algorithms.

3 PGAS Programming Model: An Overview

PGAS programming model uses fine-grain programming style that makes application performance modelling and analysis a challenging task. Recall that unlike coarse-grain programming model where large amounts of computational work are done between communication events, fine-grain programming style relatively small amounts of computational work can be done between communication events. Therefore, if the granularity is too fine it is possible that the overhead required for communications takes longer than the computation. PGAS programming model provides two major characteristics namely data partition and the locality that should be used to reduce the communication overhead and get better performance. UPC is one of partitioned global address space programming language based on C and extended with global pointers and data distribution declarations for shared data [1].

A PGAS-based algorithm depends on the number of threads and how they are accessing the space [1], [2]. A number of threads can work independently in a Simple Program Multiple Data (SPMD) model. Threads communicate through the shared memory and can access shared data while a private object may be accessed only by its own thread. The PGAS memory model, used in UPC for example, supports three different kinds of pointers: private pointers pointing to the shared address space, pointers living in shared space that also point to shared data, and private pointers pointing to data in the thread's own private space. The speed of local shared memory accesses will be slower than that of private accesses due to the extra overhead of shared-to-local address translation [5]. Also, remote accesses in turn are significantly slower because of the network overhead and address translation process.

According to these PGAS programming features, the fine-grained programming model is simple and easy to use. Programmers need to only specify the data to be distributed across threads and reference them through special global pointers. However, a PGAS program/algorithm can be considered efficient (compared to another algorithm) if it achieves the following objectives [2]. The first objective is to minimize the inter threads communication overhead incurred by remote memory accesses to shared space. The second objective is to maximize the efficient use of parallel threads and data placement or layout together with data locality (i.e., the tradeoff between the number of allocated threads and

the communication overhead). The third objective is to minimize the overhead caused by the synchronization points.

Recently, all efforts are focused on compiler optimization, i.e. without the control of the programmer, in order to increase the performance of the PGAS parallel programming language such as UPC. In [3] and [7], many optimization suggestions have been proposed for incorporation into UPC compilers, such as message aggregation, privatizing local shared accesses, and overlapping computation and communication to hide the latencies associated with remote shared accesses.

4 A Complexity Model for PGAS Algorithms

A PGAS algorithm consists of a number of threads that use a SPMD model to solve a given problem. Each thread has its own subset of shared data and can coordinate and communicate with the rest of threads by writing and reading remote shared objects or data to perform its computation. In addition, each thread performs local computations and memory requests to its private space. The algorithm complexity can be expressed using algorithm-dependent parameters to measure the number of basic operations and the number of read/write requests. More precisely, the algorithm complexity can be expressed using T_{comp} and T_{comm}, where T_{comp} denotes the computational complexity, and T_{comm} the number of read and write requests. The computational complexity depends on the problem size, denoted by N, and the number of threads, T. For example, in the case where all threads execute the same workload (i.e., the workload is uniform), the number of computation operations for all threads is similar. However, in irregular applications, certain threads could perform more or less computational operations. To includes the idle time induced by waiting, e.g. for other threads, we consider that T_{comp} is the maximum of T^i_{comp} for $1 \leq i \leq T$, where T is the number of threads. It is worth notice that each thread should be executed by one processor; the number of threads T is equal to the number of processors P. In order to calculate T^i_{comp}, we consider that a PGAS algorithm can be composed of a sequence of phases eventually separated by synchronization points. At a given phase, a thread can compute only or compute and communicate. Therefore, alike sequential programming, the computational complexity of a thread i in all elementary phases j, $j \in \phi$, is $T^i_{comp} = max_{j=1:\phi}(T^i_{comp}(j))$, where ϕ is the number of all elementary phases. The computational complexity T_{comp} of the algorithm is the highest computational complexity of all threads and is determined as follows: $T_{comp} = max_{i=1:T}(T^i_{comp})$. The complexity is dominated by the thread that has the maximum amount of work. Unlike the computational complexity, the communication complexity of a thread i is the sum (over all phases) of $T^i_{comm}(j)$ as follows: $T_{comm} = \sum_{j=1}^{\phi} T^i_{comm}(j)$. The communication complexity of the algorithm is the highest communication complexity overall threads and is determined as follows: $T_{comm} = max_{i=1:T}(T^i_{comm})$. T_{comm} is an upper bound on the number of memory requests made by a single thread.

The communication complexity $T^i_{comm}(j)$ at each phase j and for each thread i, depends on the number of requests to private and shared memory spaces,

called the pointer complexity [2]. The pointer complexity is defined to be the total number of pointer-based manipulations (or references) used to access data. There are three sorts of pointers in a PGAS algorithm represented by $N_p^i(j)$, the number of references to the private memory space, $N_\ell^i(j)$ the number of references to the local shared memory space, and $N_r^i(j)$ the number of references to remote shared memory spaces. Since accesses to remote shared memory spaces are more expensive than accesses to private and local shared memory spaces [1], [2], [5], $N_p^i(j)$ and $N_\ell^i(j)$ can be neglected due to their insignificance relative to the cost of remote references accesses to remote shared spaces $N_r^i(j)$. Therefore, the communication complexity depends mainly on the number of references to remote shared spaces, and hence we have: $T_{comm}^i(j) = O(N_r^i(j))$. This number of remote references (i.e., read and write requests) can be easily and directly computed from the number and layout of data structures declared and used in the algorithm. More precisely, it depends on the following parameters: the number of shared Data elements D declared and used in the algorithm, the number of Local shared data elements L to a thread ($L \leq D$), the number of threads T, and the size N of the shared Data structures . In what follows, we consider the worst case when any computational operation could equally involve global or local shared requests to shared memory spaces. Each thread also needs to write or read from other threads' partitions. Using these parameters, the number of remote references represents the communication complexity (i.e., number of read and write requests), at each phase j, as follows:

$$T_{comm}^i(j) = O(N_r^i(j)) = O(\frac{(D_j^i - L_j^i)(T-1)}{T} T_{comp}^i(j)) \tag{1}$$

$T_{comp}^i(j)$ captures the number of computational operations, at the phase j, and depends only on the average number of elements in shared data structures N and the number of threads T. D_j^i is the number of all shared pointers to remote spaces (i.e., local spaces of other threads) used in the algorithm, for the thread i in the phase j. L_j^i captures the number of shared references (i.e., pointers) to a local space of the thread i in the phase j.

Let us consider that $\alpha_j^i = \frac{D_j^i - L_j^i}{D-L}$. Equation 1, which determines the communication complexity of thread i at each phase j, can be rewritten as follows:

$$T_{comm}^i(j) = O(\frac{\alpha_j^i (D-L)(T-1)}{T} T_{comp}(j)) \tag{2}$$

To estimate the communication complexity of the algorithm, let us now consider, that a certain thread i has the highest computational complexity $T_{comp} = T_{comp}^i(k)$ overall threads and overall phases k. According to equation 2, the communication complexity of the algorithm is:

$$T_{comm}^i = \max_{i=1:T} (\sum_{j=1}^{\phi} T_{comm}^i(j)) = O(\frac{\alpha(D-L)(T-1)}{T} T_{comp}) \tag{3}$$

where $\alpha = \max_{i=1:T}(\sum_{j=1}^{\phi} \alpha_j^i)$. It should be noted here that the computation complexity depends all time on the size of the problem N and the number of

threads T, $T_{comp} = f(N, T)$. In addition, the communication complexity depends on the number of shared data elements D declared and used in the algorithm, the number of local shared data elements L to a thread, the size of the shared data structures N, the number of threads T (T is equal to the number of processors P), and the parameter α, $f(D, L, N, T, \alpha)$, where α is a function of T. For example, let us consider the following $N \times N$ matrix multiplication algorithm, $C = A \times B$:

Matrix multiplication algorithm

Input: a $N \times N$ matrix A and a $N \times N$ matrix B
Output: the $N \times N$ matrix $C = A \times B$
Data Distribution: A , B, and C are distributed in round-robin using four cases
//Loop through rows of A with affinity (e.g., i)
For $i \leftarrow 1$ and $affinity = i$ to N parallel Do
For $j \leftarrow 1$ to N Do *// Loop through columns of B*
$C(i, j) \leftarrow 0$
For $k \leftarrow 1$ to N Do *// perform the product of a row of A and a column of B*
$C(i, j) \leftarrow C(i, j) + A(i, k) \times B(k, j)$
End for k
End for j
End for i

This algorithm takes two matrix A and B as an input and produces a matrix C as an output. These matrices are $N \times N$ two-dimensional arrays with N rows and N columns. The data distribution allows us to define how these matrices will be partitioned between threads. Four cases of data distribution are considered in this example and will be described in section 5. The first loop allows the distribution of independent work across the threads, each computing a number of iterations represented by the field $affinity$. This field determines which thread executes a given iteration. In this example, the affinity i indicates that the iteration i will be performed by thread (i mod T). Using this field, iterations will be distributed across threads in round-robin manner and each thread will process only the elements that have affinity to it avoiding costly remote requests to shared spaces. This parallel loop can be translated into the parallel construct of the considered PGAS language, UPC for example [1].

In this algorithm, the computational complexity T_{comp} requires $O(\frac{N^3}{T})$ multiplication and addition operations, and the communication complexity T_{comm} requires $O(\frac{\alpha(D-L)(T-1)}{T^2}N^3)$ requests to remote shared spaces, where D=3, and $L \in \{0, 1, 2, 3\}$. The value of L depends on the case of data distribution considered (see section 5). According to this analysis, the communication complexity of a given PGAS algorithm achieves a lower bound when $L = D$, which means that threads compute on their own data without performing any communication to access remote ones. In other words, a communication complexity of a PGAS algorithm achieves its lower bound when $L = 0$; all data are accessed with remote shared pointers.

The speedup of a PGAS algorithm, denoted by S_{PGAS}, is defined as the ratio between the time taken by the most efficient sequential algorithm, denoted by

T_{seq}, to perform a task using one thread and the time, denoted by T_{par}, to perform the same task using T threads (i.e., $T = P$). This speedup can be estimated as follows:

$$S_{PGAS} = \frac{T_{seq}}{T_{par}} = O(\frac{T_{seq}}{1 + \alpha(D-L)\frac{(T-1)}{T}T_{comp}}) \quad (4)$$

Since the most optimal sequential algorithm yields to $T_{comp} \leq \frac{T_{seq}}{T}$, the speedup, given in the equation 4, can be rewritten as follows:

$$S_{PGAS} = O(\frac{T}{1 + a(T-1)}) \quad (5)$$

where $a = \frac{\alpha(D-L)}{T}$ is a key parameter that emphasizes the influence of the locality degree of the algorithm in function of the number of threads, i.e., number of processors. According to this equation, the PGAS speedup is optimal, i.e., linear, when S_{PGAS} is close to T (i.e., $S_{PGAS} = O(T)$). This linear speedup can be obtained when $D = L$; i.e., there are no remote requests to shared spaces. Therefore the designer should simultaneously maximize the locality degree L, and minimize the usage of remote shared pointers represented by α. When T approaches infinity, $a(T-1)$ is bounded by $a(D-L)$, where α is not constant, but depends on the number of threads. Hence, even if we increase the number of threads, in parallel the number of processors, to run the algorithm, the performance is limited by the impact of the number of requests to remote shared memory spaces a. Therefore, the designer of PGAS algorithms should attempt to reduce this parameter to the smallest possible value. It should be noted also that since the model considers the asymptotic complexity of the computation and the communication, the overlapping mechanism that allows the computation to proceed without waiting for the entire data transfer to be completed is also captured by the model.

5 Experimental Results

In this section, to validate the performance model presented in this paper, we have implemented a UPC program to perform matrix multiplication algorithm with different data distribution schemes. The matrix multiplication is selected because it is simple to show the effectiveness of the model. In addition, matrix multiplication represents one of the most important linear algebra operations in many scientific applications..

The experiments were done using the GCC UPC toolset running on Origin 2000. The SGI Origin 2000 platform consists of 16 processors and 16GB of SDRAM interconnected with a high speed cache coherent Non Uniform Memory Access (ccNUMA) link in hypercube architecture. The GCC UPC toolset provides a compilation and execution environment for programs written in UPC language.

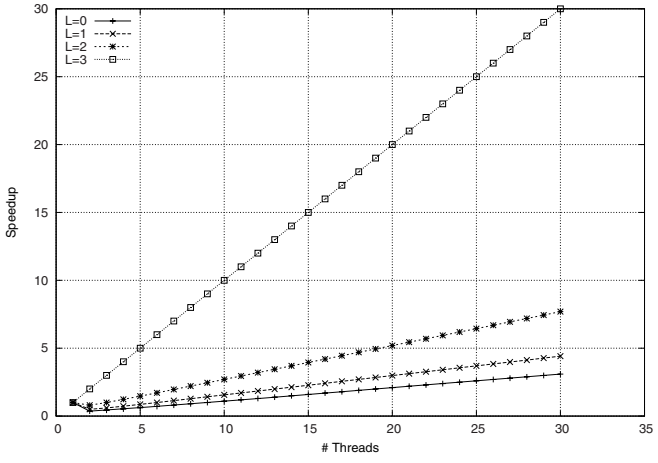

Fig. 1. The theoretical speedup vs. L

In this experimentation, three data structures A, B, and C ($D = 3$) are considered. We consider the case where all matrices are of size ($N = 128$) and the number α is equal to 3. Four different data distribution cases ($L = 0$, $L = 1$, $L = 2$, and $L = 3$) are also considered in these experiments. In the first case, $L = 0$, matrices A, B, C are partitioned among T threads (i.e., processors) by assigning, in round robin, each thread $\frac{N}{T}$ columns of A, B, and C respectively. More precisely, elements of these matrices will be distributed acroses the threads element-by-element in round-robin fashion. For example, the first element of the matrix A is created in the shared space that has affinity to thread 0, the second element in the shared space that has affinity to thread 1, and so on. In the second case, $L = 1$, the compiler partitions computation among T threads by assigning each thread $\frac{N}{T}$ rows of A and $\frac{N}{T}$ columns of B and C. More precisely, the elements of matrices A and B are distributed element-by-element in round-robin fashion, i.e., each thread gets one column in round robin fashion. At the end, each thread up with $\frac{N}{T}$ columns of B and C, where (N mod $T = 0$). The elements of the matrix A will be distributed across the threads N-elements by N-elements in round-robin fashion. In the case where $L = 2$, the compiler partitions computation among T threads by assigning each threads $\frac{N}{T}$ rows of A, and C, and $\frac{N}{T}$ columns of B. This case is similar to the second case except that the elements of the matrix C will be distributed across the threads N-elements by N-elements in round-robin fashion. The last case, $L = 3$, is similar to the third case with the exception that each thread has the entire B.

Recall that from the model, the designer should maximize the locality degree L, and minimize α, i.e. minimize the usage of remote shared pointers. Figure 1 presents the theoretical speedup, calculated from the model (eq. 5), in function of the locality degree, $L = 0$, $L = 1$, $L = 2$, and $L = 3$ ($\alpha = 3$). This figure shows that as we increase L, the speedup increases and tends to become linear when $D = L$, i.e., good data locality. More precisely, the algorithm (matrix multiplication) without remote communication ($L = 3$) performs better than the algorithm with

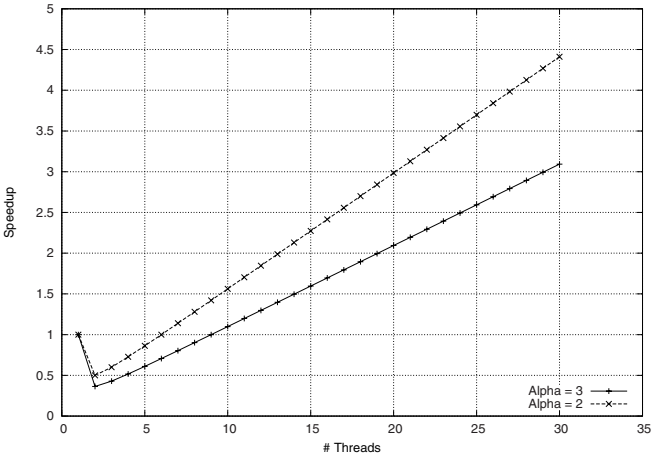

Fig. 2. The theoretical speedup vs. α

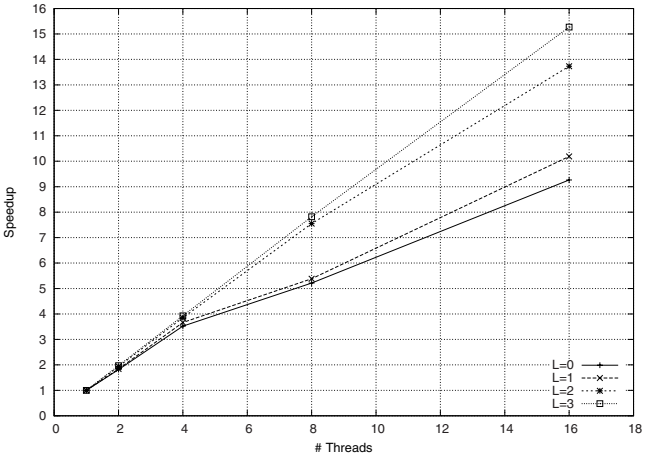

Fig. 3. The experimental speedup vs. L

a minor remote communication ($L = 2$) that performs better than the algorithm with an intermediate remote communication ($L = 1$). This last case performs better than the algorithm with a larger remote communication ($L = 0$).

According to the model, the designer should also attempt to reduce the parameter α to the smallest possible value. To illustrate this point, let us consider the matrix multiplication algorithm with three data structures ($D = 3$), and L is equal to 0. Figure 2 presents the theoretical speedup in function of the number α. This figure shows that as we decrease the the usage of remote shared pointers (i.e., from $\alpha = 3$ to $\alpha = 2$), the speedup increases.

The objective of the experimentation is to prove this statement, which is given from the complexity model. Each thread is executed in one processor. The

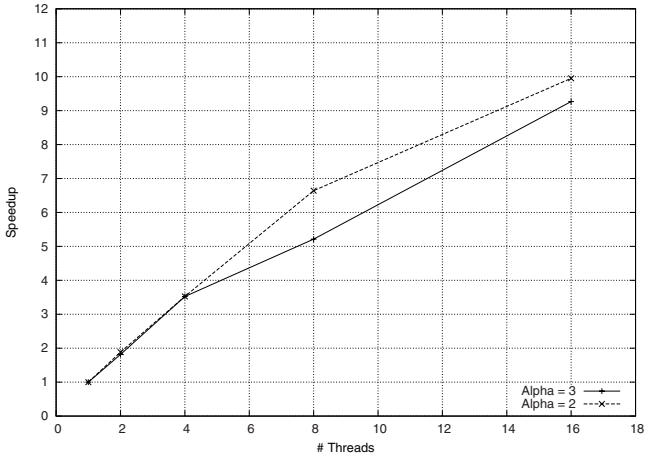

Fig. 4. The experimental speedup vs. α

results depicted in figure 3 illustrate that as we increase L, the speedup increase to become linear when $L = D$. We can see also in figure 4 that decreasing the usage of remote shared pointers (i.e., from $\alpha = 3$ to $\alpha = 2$), the speedup increases. It should be noted that the objective of these experiments is not to compare the performance of this program according to the literature but to show the following behavior: as we increase L and decrease α there is an increase in the speedup.

These primary experimental results corroborate the result obtained from the complexity model described above and show that to improve the algorithms' performance, by decreasing the communication cost, the programmer must increase as much as possible the value of the locality parameter L and minimize the usage of remote shared pointers represented by α.

6 Conclusion and Future Work

In this paper, we present a performance model based on the complexity analysis that allows programmers to design and analyze their algorithms independent of the target platform architecture. Given an algorithm, we have shown that we can extract the program-dependent parameters to calculate the computation and the communication complexity. According to this model, to obtain algorithms that perform well, the remote communication overhead has to be minimized. The choice of a good data distribution is of course the first step to reduce the number of requests to remote shared spaces. Consequently, the programmer is able to ensure good data locality and obtain better performance using this methodology as a tool to design better algorithms. The utility of this model was demonstrated through experimentation using matrix multiplication program under different data distribution schemes. According to the complexity model and the experimental results, the most important parameters are the number D and the size N

of data structures, the locality degree L, the number α used in the algorithm, and the number of threads. Therefore, the algorithm designer should simultaneously minimize D and α, and maximize L to get better performance.

Future work addresses additional experiments with larger applications and other platforms. The objective is to provide a tool for the comparison of alternate algorithms to the same application without necessarily resorting to an actual implementation. In addition, we will extend this complexity model to predict the computational and the communication times by including machine-dependent parameters such as the latency and the overhead.

References

1. El-Ghazawi, T., Carlson, W., Sterling, T., Yelick, K.: UPC: Distributed Shared Memory Programming. Book. John Wiley and Sons Inc., New York (2005)
2. Gaber, J.: Complexity Measure Approach for Partitioned Shared Memory Model, Application to UPC. Research report RR-10-04. Universite de Technologie de Belfort-Montbeliard (2004)
3. Cantonnet, F., El Ghazawi, T., Lorenz, P., Gaber, J.: Fast Address Translation Techniques for Distributed Shared Memory Compilers. In: International Parallel and Distributed Processing Symposium IPDPS 2006 (2006)
4. Cameron, K.W., Sun, X.-H.: Quantifying Locality Effect in Data Access Delay: Memory logP. In: IPDPS 2003. Proc. of the 17th International Symposium on Parallel and Distributed Processing, p. 48.2 (2003)
5. Zhang, S., Seidel, R.Z.: A performance model for fine-grain accesses in UPC. In: 20th International Parallel and Distributed Processing Symposium, p. 10 (2006) ISBN: 1-4244-0054-6
6. Juurlink Ben, H.H., Wijshoff Harry, A.G.: A quantitative comparison of parallel computation models. In: Proc. 8th ACM Symp. on Parallel Algorithms and Architectures, pp. 13–24. ACM Press, New York (1996)
7. Chen, W-Y., Bonachea, D., Duell, J., Husbands, P., Iancu, C., Yelick, K.: A Performance Analysis of the Berkley UPC Compiler. In: Annual International Conference on Supercomputing (ICS) (2003)
8. Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., von Eicken, T.: LogP: Towards a Realistic Model of Parallel Computation. In: PPOPP 1993: ACM SIGPLAN, pp. 1–12. ACM Press, New York (1993)
9. Gerbessiotis, A., Valiant, L.: Direct Bulk-Synchronous Parallel Algorithms. J. of Parallel and Distributed Computing 22, 251–267 (1994)
10. Gibbons, P.B., Mattias, Y., Ramachandran, V.: Can a Shared-Memory Model Serve as a Bridging Model for Parallel Computation? In: SPAA 1997. 9th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 72–83. ACM Press, New York (1997)
11. Valiant, L.G.: A Bridging Model for Parallel Computation. Comm. of the ACM 33(8), 103–111 (1990)
12. Maggs, B.M, Matheson, L.R., Tarjan, R.E.: Models of Parallel Computation: A Survey and Synthesis. In: Proceeding of the Twenty-Eight Hawaii Conference on System Sciences, vol. 2, pp. 61–70 (1995)