Evaluation of UPC Programmability Using Classroom Studies

Carlos Teijeiro, Guillermo L. Taboada, Juan Touriño, Basilio B. Fraguela, Ramón Doallo
Computer Architecture Group
University of A Coruña, Spain
{cteijeiro,taboada,juan,basilio,doallo}@udc.es

Damián A. Mallón, Andrés Gómez, J. Carlos Mouriño
Galicia Supercomputing Center
Santiago de Compostela, Spain
{dalvarez,agomez,jmourino}@cesga.es

Brian Wibecan
Hewlett-Packard
Nashua (NH), USA
brian.wibecan@hp.com

***Abstract*—The study of a language in terms of programmability is a very interesting issue in parallel programming. Traditional approaches in this field have studied different methods, such as the number of Lines of Code or the analysis of programs, in order to prove the benefits of using a paradigm compared to another. Nevertheless, these methods usually focus only on code analysis, without giving much importance to the conditions of the development process and even to the learning stage, or the benefits and disadvantages of the language reported by the programmers. In this paper we present a methodology to accomplish a programmability study with UPC (Unified Parallel C) through the use of classroom studies with a group of novice UPC programmers. This work will show the design of these sessions and the analysis of the results obtained (code analysis and survey responses). Thus, it is possible to characterize the current benefits and disadvantages of UPC, as well as to report some desirable features that could be included in this language standard.**

I. Introduction

In order to measure the benefits of using UPC, a PGAS (Partitioned Global Address Space) extension to C, for parallel programming, a specific methodology based on classroom studies has been used. This work presents the results of two classroom studies, which consist of four-hour sessions with a group of UPC-unexperienced programmers organized in different stages. First, the participants fill out a form to characterize their profile. Then, a seminar explaining the basic constructs of UPC (using a slideshow and some practical examples) is given to the programmers. Afterwards they are asked to parallelize several sequential codes in order to test the acquired skills. Finally data about their impressions on UPC and some detected benefits and disadvantages are obtained. The main advantages of this approach are (1) the time control of the development stage, (2) the use of unexperienced programmers in the studied language but with some general background knowledge on different programming paradigms, and (3) the inclusion of their opinions as a complement to the analysis of the developed codes, which gives some guidelines to identify desirable features in a parallel programming language. This information is very useful for future UPC programmability improvements. In the first session, the participants are final year students of the B.S. in Computer Science Engineering [1] at University of A Coruña (UDC), whereas in the second one the participants are a heterogeneous group of research staff at the Galicia Supercomputing Center (CESGA) [2]. These two groups present special features that are interesting for the analysis, specially in terms of programmer profile. The students at UDC are a quite homogeneous group, with minor variations in their academic curricula, but the staff at CESGA present clearly different profiles, as they have different degrees from different universities.

This paper is organized as follows. First, some general remarks on programmability in High Performance Computing (HPC) are made, and the related work on UPC in this field is presented. Then, the design of the activities is explained, and the codes and software used in the sessions are presented. Afterwards, detailed information about the most relevant results is given. Finally, some conclusions are extracted from this study.

II. Programmability in HPC

In the last years, several works on programmability and productivity for HPC have been developed. The most important results on this area are related to the High Productivity Computer Systems (HPCS) project [3], funded by DARPA, which led to the proposal of three relevant languages that focus on programmability (X10, Chapel and Fortress) [4]. These languages have been designed specifically to improve programmability and productivity in code development, but they are not mature enough yet. Some other languages, such as the PGAS-based UPC, Co-Array Fortran or Titanium, have been designed as extensions of well-known programming languages (C, Fortran and Java, respectively) to provide parallel programming by means of the addition of syntactic constructs.

Many studies related with this subject are usually devoted to general considerations and requirements for a parallel language in terms of productivity [5], as well as comments about benefits and disadvantages of the most popular approaches (MPI and OpenMP). Moreover, there are also some works on programmability in HPC devoted to the proposal and analysis of different metrics [6], [7] and the design of specific benchmarks [8]. An interesting conclusion from some of these studies is that a language is considered to be good in terms of programmability if it contains expressive constructs which

allow a more compact and simple coding, hence making low-level complexity transparent to the user. However, programming languages are difficult to compare in terms of real programmability, because programmability-oriented languages are quite novel and code developers are used to working with the most popular approaches. Therefore, the success of these new languages for HPC seems to be bound only to their productivity enhancements, whereas extensions of traditional languages can benefit from a more broad acceptance.

Additionally, a good way to prove if a language provides good programmability is to make a survey on a group of programmers. The benefits and disadvantages reported by various code developers, especially when they have different skills, can give valuable information about the programming language and also help guess if it could become popular among the parallel programming community.

III. Related Work

Until now, most studies on UPC have focused on performance, and there are still very few specific works on UPC related to programmability. The most relevant previous works deal with constructs and algorithms focused on performance increase [9], [10] compared to other approaches measuring programming effort in terms of Lines Of Code (LOC) [11].

Classroom programmability sessions are used as a good reference in order to measure productivity. Some tests with homogeneous groups of students have also been carried out, obtaining general conclusions for different languages and paradigms [12], [13], [14]. UPC has also been considered for a programmability study [15] in a comparison with MPI that includes an analysis of statistical significance of the results. However, the development times were not taken into account and the experimental conditions of the study were undefined.

In these studies, the typical measures that are used to evaluate programmability are Source LOC (SLOC) and speedup, directly measured from the codes developed during a programmability session. Here, special applications to manage log and report complete information about the work performed by each participant in the study are generally used [16]. Among these tools it is worth mentioning UMDInst [17], which consists of a set of wrappers that create XML logs including the most relevant actions performed at code development (edition, compilation and execution), also saving snapshots of the codes. These features help to give a more accurate measure of the development time and cost associated to the parallelization of a code.

IV. Programmability Study

This work presents an improved methodology that addresses concerns associated with previous studies of programmability. In this study we (1) control development time, (2) provide an assessment of the profile of the programmers, and (3) conduct a survey of impressions of the language. We gathered programmability results from two different environments. The classroom studies are closed sessions with a restricted and well-defined framework to accomplish a programmability evaluation. These restricted classroom studies allow to control the duration and keep track of the whole development process, as well as to ensure that all participants have the same initial and basic knowledge of UPC (presented at the beginning of the session). Control on the information flow is performed by using logging tools.

The two sessions organized at UDC and CESGA have followed the same overall structure. First, every participant was asked to fill out an initial questionnaire about his academic profile and parallel programming background, as well as his interest on this area. The questions have been adapted to the participants of each study.

After the initial test, the participants attended a seminar on UPC. The contents of this talk were taken from slides used in UPC seminars at UC Berkeley [18], and included all basic concepts and constructs needed to understand the PGAS paradigm (shared address space vs. private address space) and develop UPC codes (upc_forall construct, barrier synchronizations, pointers, distribution of shared arrays and raw memory copies). During this talk, the students were asked to test some sample codes (Hello World, Pi Computation using the Monte Carlo Approach and Matrix-Vector Multiplication). The UPC seminar, including the explanations and the execution of the test codes, lasted about 1 hour and 30 minutes.

Then, at the development stage, the participants were asked to develop three parallel codes in UPC from their sequential versions, implemented in C, that were given to them. The overall coding time was 2 hours. The three proposed codes are:

- A simple Stencil operation on a 10^6-element vector, analogous to one of the example codes included in the Berkeley UPC distribution. The vector has 10^6 elements and the operation is performed 100 times (it uses an external 100-iteration loop).
- The Buffon-Laplace Needle problem, a Monte Carlo simulation that gives an accurate approximation of Pi based on the probability that a needle of length l that is dropped in a grid of equally spaced parallel lines will touch at least one line. The number of predefined trials is 10^7.
- The Computation of the Minimum Distance among different nodes in a graph (a version of the Floyd-Warshall algorithm [19]). This code was implemented by some of the students at UDC as an MPI project during a previous course on parallel programming. The size of the source distance matrix is 500x500 (that is, 500 nodes).

After parallelizing each code, the students were asked to run their codes and then report their performance. Finally, the participants had to fill out a final survey about their impressions about UPC, their interest on the language, the benefits or disadvantages they could notice and the features they would like to see in UPC.

The number of participants that attended the classroom study at UDC were 22 final year students of the B.S. in Computer Science Engineering at UDC with some previous knowledge of MPI and OpenMP. At CESGA, 13 programmers

with different profiles (e.g. B.S. degrees and PhD in computer science, physics and mathematics) took part in the experiment. In general, all participants at CESGA had experience with programming languages, but very few reported to have a previous knowledge on parallel programming. Even some of these programmers did not have much experience with C, as they were used to working with other languages (e.g. Java and PHP).

The experimental testbed for the tests consisted of an 8-node InfiniBand cluster, with 4 cores per node and hyperthreading [20], and 16 single-core nodes of a Myrinet cluster [21]. All performance results were obtained using the Berkeley UPC compiler, version 2.8.0 [22]. The UMDInst system has been used in the cluster to generate logs. A summary of the most relevant results from each code has been obtained with two Perl scripts that parse the UMDInst logs. Additionally, the number of SLOCs for each code has been got using the CLOC Perl script [23].

V. Analysis of Results

The analysis of the different results obtained in the two classroom studies has been accomplished using two sources of information: the codes developed by each participant and their profiles and opinions about UPC. The study of the codes is based on the speedup achieved, the number of SLOCs and the development time for each code.

A. Overall Summary

Table I presents a summary of the UPC codes developed in the classroom studies undertaken, indicating the number of SLOCs of the sequential versions of the three codes (in order to be used as a reference for the SLOC analysis) and the number of correct and incorrect codes developed by the participants. When the sum of the correct and incorrect versions of a code is not equal to the number of participants in the session, it indicates that some participants in that session have not even started developing a code. Two conclusions can be extracted from this table: on the one hand, most of the participants could obtain a correct solution for the Stencil and the Buffon-Laplace Needle problems; on the other hand, few of them were able to parallelize the Minimum Distance Computation.

Regarding the incorrect implementations obtained by some of the participants, there are several points in common among them. The Stencil code is quite simple, but four people (2 at UDC and 2 at CESGA) did not obtain a correct solution, because all of them parallelized the external iterations loop instead of the stencil loop. Although the results obtained with these codes are correct, this work distribution causes all threads to perform all the operations over the whole array instead of splitting the array processing among the different threads. Thus, these participants have misunderstood the basic concepts of SPMD processing and work distribution with `upc_forall`, and hence their implementations for this code are highly inefficient.

The incorrect Buffon-Laplace Needle codes (7 at UDC and 3 at CESGA) also shared a common error: these participants forgot to synchronize the threads after the computation of the Pi estimation in each thread. This race condition can produce an erroneous result, because there is no guarantee that the partial results are updated when the root thread tries to get them from shared memory. Additionally, 6 of these 10 erroneous codes also used a scalar shared variable to get the partial results for every thread, instead of storing them in a shared array and performing a reduction on it, thus causing a race condition. Again, this error is due to a misunderstanding of the UPC memory model.

Moreover, unlike in the previous cases, the characterization of the errors in the Minimum Distance code is more difficult, because many of them are due to a bad initialization of the matrix or some misunderstanding of the algorithm. In general, many participants (even the ones that developed a correct code) found it difficult to deal with shared array arguments in functions because of the block size definitions.

The results of each session are analyzed in more detail in the following sections. Tables II-VII show the results of all correct UPC codes developed in the programmability session at UDC (Tables II-IV) and CESGA (Tables V-VII), respectively. The codes have been classified according to the performance obtained in terms of speedup when they are executed with up to 16 threads. The speedup is specified qualitatively, and each of these values corresponds to a different behavior of the speedup for this code when the number of threads increases (e.g. if the efficiency is almost 100%, the speedup is considered as "excellent"). Alongside the speedup, the average (μ) SLOCs and the average development time (Dev. Time) of all the codes included in each group are shown in every table, because they can help give a measure of the acquired capability of each group to develop UPC programs.

B. Programmability Session at UDC

The results presented in Table II show that there are five possible classifications in terms of speedup for the 20 UPC Stencil codes developed in the session at UDC. As the parallelization of this code is defined using a few syntactic constructs, it is easy to find a correlation between the use of these constructs and the runtime speedup obtained by each code. Thus, the only one that included the correct parallel constructs, and also used a privatization method for one of the arrays, could obtain an "excellent" speedup (which means that it was very close to the ideal). A "quite good" speedup (about 80% of parallel efficiency) has been obtained by codes that included the correct parallel constructs, but without a privatization method (that is, they used a `upc_forall` loop with an optimal blocking factor for both arrays and performed all operations in shared memory), thus obtaining a slightly lower speedup than with privatizations. Most of the Stencil codes obtained a speedup rating of "good", because they were parallelized using the `upc_forall` loop, but with non-optimal array blocking for both arrays. An "average" speedup has been obtained by two codes that implemented the `upc_forall` loop, but

Table I
SUMMARY OF CODES OBTAINED IN THE CLASSROOM STUDIES

Code	# SLOCS (seq)	# Participants	# Correct	# Incorrect
Stencil (UDC)	21	22	20	2
Stencil (CESGA)	21	13	10	3
Buffon-Laplace N. (UDC)	90	22	15	7
Buffon-Laplace N. (CESGA)	90	13	9	3
Minimum Dist. (UDC)	63	22	7	8
Minimum Dist. (CESGA)	63	13	2	4

performed a particular array blocking: the two arrays used for the Stencil operation had a different blocking factor, that is, one had a block distribution and the other used a cyclic distribution. Thus, depending on the definition of affinity in the `upc_forall` loop, both codes achieve different speedup, but it is always lower than in the previous cases. Finally, one code could not get any speedup at all, because of a bad definition of the affinity in the `upc_forall` loop, which maximized the number of remote accesses.

Additionally, the study of SLOCs for Stencil indicates that a quite good speedup can be obtained without increasing the size of the code, but the best performance is achieved with more lines. This is due to the use of privatizations, that requires additional processing (e.g. the definition of private arrays and the copy of the shared arrays to private memory).

In terms of development time, the participants at UDC spent around 50 minutes on average with this code. As stated during the session, many of them had reviewed all the information about UPC that was given to them during the seminar in order to develop their first UPC code, therefore some of this development time can be assigned to a small learning curve. Nevertheless, some significant differences among participants were appreciated here.

Table II
STENCIL WITH 10^6 ELEMENTS (UDC)

Speedup	# Codes	μ # SLOCs	μ Dev. Time
Excellent	1	28	38' 19"
Quite good	4	22	49' 35"
Good	12	23	50' 28"
Average	2	22	50' 20"
Bad	1	23	1 h 1' 43"

Table III presents the results for the Buffon-Laplace Needle code. All the developed codes achieve excellent speedup. The reason is that Buffon-Laplace Needle presents few parallel alternatives: a correct parallel code is likely to obtain high speedup, and the lack of a feature in the code (e.g. synchronization barriers, shared array definition) tends to result in an incorrect program.

It is also significant that the amount of development time for this second exercise is less than for the first one. This happens because this code is based in the evaluation of trials similarly to the computation of Pi using the Monte Carlo method, which was proposed as an example in the UPC seminar. Thus, many participants probably found the analogy between these two codes and they could obtain a correct code easily. Regarding the length of the codes, the average number of additional SLOCs used here in order to parallelize this code is 8.

Table III
BUFFON-LAPLACE NEEDLE WITH 10^7 TRIALS (UDC)

Speedup	# Codes	μ # SLOCs	μ Dev. Time
Excellent	15	98	28' 43"

Table IV shows the results of correct Minimum Distance codes, where the speedups are classified as "excellent" (4) and "bad" (3). The use of correct `upc_forall` loops and privatizations of variables is the reason why the best four versions of this code obtained good speedup.

In terms of SLOCs, privatizations imply the inclusion of some additional lines of code (in the best four cases, 12 lines on average). Other improvements used less SLOCs (on average, 6 were used in different parallelizations using `upc_forall` loops), but the benefits were not so important in terms of speedup.

The mean development time for the best four codes is high, probably because the correct implementation of privatizations for these codes may have taken longer.

Table IV
MINIMUM DISTANCE COMPUTATION WITH 500 NODES (UDC)

Speedup	# Codes	μ # SLOCs	μ Dev. Time
Excellent	4	75	1 h 1' 34"
Bad	3	70	46' 26"

C. Programmability Session at CESGA

Table V presents the speedup classification of the Stencil codes developed in the classroom study at CESGA. Here there are only two types of codes: the ones that achieved a quite good speedup and the ones that showed bad speedup. The first ones used a correct `upc_forall` loop and a suitable blocking of arrays, whereas the latter used a wrong definition of the affinity in the `upc_forall` loop for the selected array blocking.

The average number of SLOCs for these codes (22 and 24 SLOCs) is very close to the sequential code (21 SLOCs). However, there are significant differences among the codes developed at CESGA, which are due to the different profiles of the participants.

The development time of this code is also a bit lower than at UDC, because many participants did not spend too much time testing this code, and some of them had a great ability to quickly develop the required code.

Table V
STENCIL WITH 10^6 ELEMENTS (CESGA)

Speedup	# Codes	μ # SLOCs	μ Dev. Time
Quite good	7	24	40' 19"
Bad	3	22	48' 30"

The results shown in Table VI are analogous to the ones obtained at UDC: all the correct codes achieved the best possible speedups. Nevertheless, there are significant differences in terms of SLOCs and development time, as the participants at CESGA used, on average, more SLOCs and more time to develop the parallel code.

Once again, there are noticeable differences among CESGA programmers, because many of them did not realize quickly that this program was analogous to the Pi computation presented in the seminar, and therefore they had problems on deciding which was the strategy to parallelize this code. This led to a mean number of editions and compilations higher than in the UDC session. However, as with Stencil, some participants could get to parallelize this code quite quickly.

Table VI
BUFFON-LAPLACE NEEDLE WITH 10^7 TRIALS (CESGA)

Speedup	# Codes	μ # SLOCs	μ Dev. Time
Excellent	9	101	46' 20"

The Minimum Distance code has posed a great challenge to CESGA programmers: this code was not familiar to any of them, unlike in the UDC session. Therefore, its complexity and the time involved in developing a correct solution to the previous codes were the reasons why only half of the participants in the session at CESGA started developing the parallel version of this code. As shown in Table VII, none of the two correct codes could get a good speedup. However, it can be seen that one code obtained a slightly better speedup with a higher development time.

Table VII
MINIMUM DISTANCE COMPUTATION WITH 500 NODES (CESGA)

Speedup	# Codes	μ # SLOCs	μ Dev. Time
Bad	1	90	1 h 10' 53"
Very bad	1	66	30' 31"

D. Programmability Analysis of UPC vs. MPI

As commented before, some UDC students that attended this session had also developed previously an MPI version of the Minimum Distance code, after a 12-hour course on basic notions of parallel programming (4 hours) and MPI (8 hours). In order to present a comparison among these two codes, Table VIII shows the speedups ("+" means "excellent", "-" means "bad"), SLOCs and development time of pairs of MPI and UPC codes developed by the same students (named U-xx). Five of the seven students that obtained a correct UPC version of this code had previously developed an MPI implementation, thus these five codes will be studied in this section. The MPI development times are an estimation of the time consumed in the study and parallelization of this code, and it was reported by the corresponding student.

Table VIII
MINIMUM DISTANCE COMPUTATION - MPI VS. UPC

ID	Speedup		# SLOCs		Dev. Time	
	MPI	UPC	MPI	UPC	MPI	UPC
U-04	+	-	136	68	24 h	35'
U-05	+	+	139	73	36 h	1 h 21'
U-15	+	-	158	76	15 h	1 h 6'
U-17	+	+	159	87	15 h	1 h 31'
U-20	+	+	174	66	18 h	1 h 24'

The analysis of these UPC and MPI results clearly indicates that UPC allows an easier and faster parallelization than MPI: compared to their MPI codes, three students could get a similar speedup with UPC using less development time and SLOCs (the sequential code has 63 SLOCs). Although MPI time measurements have not been strictly controlled, the development time estimation and the SLOC count suggest that there is a high difference in terms of programmability among both approaches: the reason is that another two students that had not developed the MPI version of Minimum Distance were able to obtain a correct UPC parallel code during the classroom study (one of them with a very good speedup), which may confirm that the time necessary to understand the problem is not very high. Moreover, the learning time for MPI was longer than for UPC, which confirms the effectiveness of the latter for a quick parallelization. However, for most of the students, MPI was the first time they approached parallel programming, so this fact has to be taken into account. It is also important to note that the average development time of the best UPC codes is near an hour and a half, therefore if the time for the session were longer probably more students could have developed the UPC version of this code.

E. Reported Comments on UPC

The comments of the participants of this study on UPC are presented in Figures 1 to 3. Figure 1 indicates that the most important benefit for many of the participants is that UPC is an extension of C. Additionally, about half of the answers to the test considered that the UPC memory model is a helpful way to develop codes. Nevertheless, some differences among the two groups of participants are stated when asked whether UPC allows easy prototyping: UDC students consider that UPC facilitates prototyping, but CESGA programmers do not seem to consider this as helpful. This is probably due to their profiles: some of the CESGA programmers are used to object-oriented languages (e.g. Java) and found it difficult to

work with C, whereas UDC students are more used to working with different paradigms.

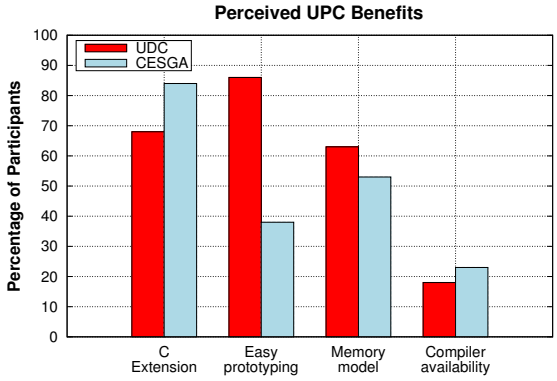

Figure 1. Reported benefits of UPC

The main drawbacks reported in Figure 2 are also different depending on the group of people studied. Both groups agree that pointer management is difficult, but also the general impression at CESGA is that UPC is a quite low-level language that may only be used in HPC. Nevertheless, participants at UDC found that the definition of block sizes was one of the most important drawbacks in UPC, alongside with the perception of poorer performance.

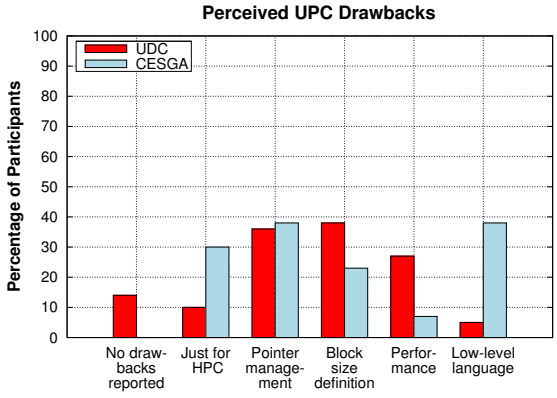

Figure 2. Reported drawbacks of UPC

The opinions about the most urgent issue to solve in UPC, that are shown in Figure 3, have followed are similar to the previous ones: CESGA programmers miss language abstractions that help to obtain a simpler code, whereas UDC students' complaints focus on language performance, as well as on general data distribution issues. Moreover, the percentage of participants at CESGA that suggested the use of language abstractions is a only bit lower than the sum of percentages of all reported issues at UDC.

These three graphs reveal that there are significantly different opinions reported by UDC and CESGA participants, and the tendencies shown in each figure are also consistent with

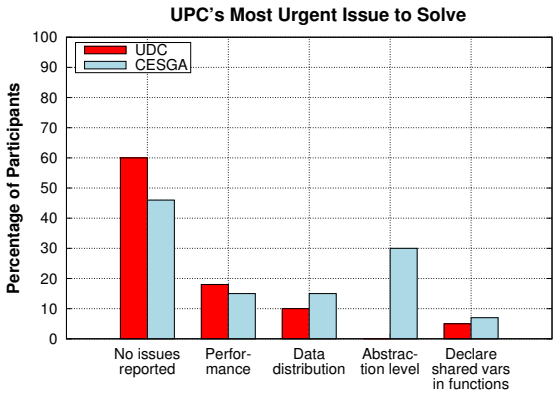

Figure 3. Most urgent reported issue to solve

the results presented in the rest of the figures. As a result, some facts that appear to be determinant in the way the participants accomplished the development UPC codes are their background knowledge, interests and motivation: participants with some specific knowledge on parallel programming have seen some advantages in terms of programmability in the use of UPC, but unexperienced participants have found it difficult to understand the implementation of parallelism in a program.

VI. Conclusions

This paper has presented the classroom study with novice UPC programmers as a method to assess the programmability of UPC. This restricted environment allows much better control of the coding time and the development problems that may occur, and the retrieved information is better than the one obtained only from the study of source codes. Additionally, the objective information acquired through this study is complemented with the opinions of the participants, which provides different points of view about the language that is being studied.

Two programmability sessions with UPC using different groups of participants have been accomplished using this methodology. Regarding the codes obtained, most of the participants could get to parallelize at least one code. The participants at UDC took advantage of their knowledge on parallel programming, and some of them also benefited from the development of the Minimum Distance code in MPI, because they were acquainted with it. This fact allowed a programmability comparison between UPC and MPI, which showed that the development time and SLOCs to parallelize the Minimum Distance code with UPC is clearly lower compared to MPI. The participants at CESGA had more difficulties in learning UPC, because many of them had no experience in parallel programming and had to learn some basic concepts (e.g. the SPMD model).

Different development problems have been encountered in the codes obtained in both sessions, but many of them were typical errors in PGAS novice programmers or were due to the design of the closed sessions, and not directly related to the

programmers' background knowledge. In fact, the difficulties in code development may also be due to the restricted time of both sessions: although the Stencil and Buffon-Laplace Needle problems are quite simple, the Minimum Distance code requires a more detailed study to develop a parallel version. However, despite these facts, the results obtained with both classroom studies were satisfactory, and some participants could even obtain UPC implementations of Stencil and Buffon-Laplace Needle that achieved good speedup using a similar number of SLOCs to the sequential versions of each code.

UPC facilitated an easier parallelization for many participants, but their profiles, background knowledge and motivation have shown to have much influence on their opinion about the language. UDC students, that were used to parallel programming, were willing to adapt to UPC and reported few disadvantages regarding the language specification. Nevertheless, CESGA programmers interpreted UPC as a low-level language specifically designed for HPC that might have some problems to become popular for a wider range of applications, especially because of the lack of high-level abstractions to provide easier programming. These different points of view also define differences in the motivation of both groups when dealing with the proposed problems, but they help to give an overall vision of the possibilities of use of UPC in different areas.

Thus, a general conclusion is that UPC can be used to develop parallel codes easier than other parallel programming approaches, but the development of libraries that focus on programmability can be interesting in order to abstract the relatively low-level C syntax and expand the use of UPC beyond the limits of HPC.

ACKNOWLEDGMENTS

This work was funded by Hewlett-Packard and partially supported by the Spanish Government under Project TIN2007-67537-C03-02 and by the Galician Government under project INCITE08PXIB105161PR. We gratefully thank Jim Bovay at HP for his valuable support, and CESGA for providing access to the SVG cluster. Also, we greatly appreciate the collaboration of the UDC students and the staff at CESGA.

REFERENCES

[1] "Computer Science Engineer - Syllabus at University of A Coruña," http://www.udc.es/estudos/en/planes/614111.asp [Last visited: August 2009].

[2] "Galicia Supercomputing Center (CESGA)," http://www.cesga.es/, [Last visited: August 2009].

[3] "High Productivity Computer Systems," http://www.highproductivity.org [Last visited: August 2009].

[4] "HPCS Language Project (HPLS)," http://hpls.lbl.gov/ [Last visited: August 2009].

[5] E. Lusk and K. Yelick, "Languages for High-Productivity Computing: the DARPA HPCS Language Project," *Parallel Processing Letters*, vol. 17, no. 1, pp. 89–102, 2007.

[6] J. Kepner, "High Performance Computing Productivity Model Synthesis," *Intl. Journal of High Performance Computer Applications*, vol. 18, no. 4, pp. 505–516, 2004.

[7] M. Snir and D. Bader, "A Framework for Measuring Supercomputer Productivity," *Intl. Journal of High Performance Computer Applications*, vol. 18, no. 4, pp. 417–432, 2004.

[8] J. Gustafson, "Purpose-Based Benchmarks," *Intl. Journal of High Performance Computing Applications*, vol. 18, no. 4, pp. 475–487, 2004.

[9] K. Yelick et al., "Productivity and Performance Using Partitioned Global Address Space Languages," in *Intl. Workshop on Parallel Symbolic Computation (PASCO'07), London, Ontario (Canada)*, 2007, pp. 24–32.

[10] R. Nishtala, G. Almasi, and C. Cascaval, "Performance without Pain = Productivity, Data Layouts and Collectives in UPC," in *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08), Salt Lake City (UT)*, 2008, pp. 99–110.

[11] F. Cantonnet, Y. Yao, M. M. Zahran, and T. A. El-Ghazawi, "Productivity Analysis of the UPC Language," in *18th Intl. Parallel and Distributed Processing Symposium (IPDPS'04), Santa Fe (NM)*, 2004, p. 254a.

[12] R. Alameh, N. Zazwork, and J. Hollingsworth, "Performance Measurement of Novice HPC Programmers Code," in *3rd Intl. Workshop on Software Engineering for High Performance Computing Applications (SE-HPC'07), Minneapolis (MS)*, 2007, pp. 3–8.

[13] A. Funk, V. Basili, L. Hochstein, and J. Kepner, "Application of a Development Time Productivity Metric to Parallel Software Development," in *2nd Intl. Workshop on Software Engineering for High Performance Computing System Applications (SE-HPCS'05), St. Louis (MO)*, 2005, pp. 8–12.

[14] J. Manzano, Y. Zhang, and G. Gao, "P3I: The Delaware Programmability, Productivity and Proficiency Inquiry," in *2nd Intl. Workshop on Software Engineering for High Performance Computing System Applications (SE-HPCS'05), St. Louis (MO)*, 2005, pp. 32–36.

[15] I. Patel and J. R. Gilbert, "An Empirical Study of the Performance and Productivity of Two Parallel Programming Models," in *22nd IEEE Intl. Symposium on Parallel and Distributed Processing (IPDPS'08), Miami (FL)*, 2008, pp. 1–7.

[16] S. Faulk, J. Gustafson, P. Johnson, A. Porter, W. Tichy, and L. Votta, "Measuring HPC Productivity," *Intl. Journal of High Performance Computing Applications*, vol. 18, no. 4, pp. 459–473, 2004.

[17] Development Time Working Group - HPCS, "UMDInst," http://hpcs.cs.umd.edu/index.php?id=18 [Last visited: August 2009].

[18] K. Yelick, "Lecture Notes on Global Address Space Programming in UPC - Applications of Parallel Computing (CS267) at UC Berkeley," http://crd.lbl.gov/~dhbailey/cs267/lecture13_upc_ky08.pdf [Last visited: August 2009].

[19] R. W. Floyd, "Algorithm 97: Shortest Path," *Commun. ACM*, vol. 5, no. 6, p. 345, 1962.

[20] "NM Cluster - Department of Electronics and Systems. University of A Coruña," http://nm.des.udc.es/, [Last visited: August 2009].

[21] "SVG Cluster - Galicia Supercomputing Center," http://www.cesga.es/content/view/409/115/lang,en/, [Last visited: August 2009].

[22] UC Berkeley / LBNL, "Berkeley Unified Parallel C (UPC) Project," http://upc.lbl.gov, [Last visited: August 2009].

[23] Northrop Grumman Corporation - IT Solutions, "CLOC - Count Lines Of Code," http://cloc.sourceforge.net/ [Last visited: August 2009].