Productivity Analysis of the UPC Language

François Cantonnet, Yiyi Yao, Mohamed Zahran and Tarek El-Ghazawi
Department of Electrical and Computer Engineering, The George Washington University
Washington, DC 20052
{ fcantonn, yyy, mzahran, tarek }@gwu.edu

Abstract

Parallel programming paradigms, over the past decade, have focused on how to harness the computational power of contemporary parallel machines. Ease of use and code development productivity, has been a secondary goal. Recently, however, there has been a growing interest in understanding the code development productivity issues and their implications for the overall time-to-solution. Unified Parallel C (UPC) is a recently developed language which has been gaining rising attention. UPC holds the promise of leveraging the ease of use of the shared memory model and the performance benefit of locality exploitation. The performance potential for UPC has been extensively studied in recent research efforts. The aim of this study, however, is to examine the impact of UPC on programmer productivity. We propose several productivity metrics and consider a wide array of high performance applications. Further, we compare UPC to the most widely used parallel programming paradigm, MPI. The results will show that UPC compares favorably with MPI in programmers productivity.

1. Introduction

The success of any programming language depends on many factors. The major one is clearly market acceptance. For a language to be widely accepted, it must combine ease of use, together with efficient execution.

The importance of ease of use is that it impacts the programmer productivity, where the productivity of the programmer in the software engineering sense depends on the program complexity and the language complexity [1]. The program complexity can be divided into three areas as follows.

Syntactic Complexity: This is the difficulty of translating between the algorithm and the code itself. A syntactically complex program is one which can be easily expressed in algorithmic way, but which becomes difficult when expressed in an actual programming language. This depends on both the application and the language used. Syntactic difficulty can make the program harder to write or to understand.

Length: While long programs could be conceptually simple, the importance of the length of code is that it measures the manual effort exerted by the programmer. Length and syntactical complexity are therefore interrelated and both contribute to the manual effort.

Conceptual/Semantic Complexity: A conceptually complex programming environment is one in which the original parallel application view is obscured as it compares to the original application view, and thus requires additional work to maintain the correspondence with the original problem. Conceptual complexity typically depends on the underlying programming model and is greatly independent of the syntax. It often results in increased execution time, as more may need to be done to make the parallel program work. An example of that would be additional communications or synchronization operations and all operations needed to manage the domain decomposition. For simplicity, we will use "manual effort" to indicate both programming and syntactical complexity introduced by a given language.

Unified Parallel C (UPC)[2] is a recently developed parallel language which has been receiving rising support and recognition by government and vendors. Many vendor and open source UPC compiler implementations now exit. UPC is a parallel extension of ANSI C, which follows a distributed shared memory programming model. In addition to having one global address space, UPC establishes an affinity between shared data and threads. This enables programmers to express locality and keep data close to the threads that are processing them. This makes UPC much easier to use than the distributed private memory models, while setting the stage for exploiting data locality and achieving high-performance. To provide these features in the best possible way, UPC has built upon the knowledge from many of its predecessors such as Split-C[3]. UPC however provides more effective memory model and synchronization techniques. Previous literature [4][5] discussed the performance issues and potential of UPC. However, the effect of UPC on the productivity has not been studied before.

As mentioned earlier, success of any programming language depends on the wide public acceptance, as well as the availability of machines that can make use of the language capabilities. UPC provides a familiar C like paradigm and many platforms support UPC. For example, UPC compilers are available now for SGI origin machines, Cray T3D/E, Compaq Alpha server, and Beowulf clusters just to name a few. Figure 1 shows the memory layout for UPC. The number of threads is given by the special constant THREADS, and each thread is identified by the special constant MYTHREAD. Each of the threads has its own private space, as well as affinity with a partition of the global shared space. This gives the ability to read and write remote memory with simple assignment statements. Furthermore, UPC enables programmers to exploit data locality by placing the data which will be used by a thread at the partition of the shared space which has affinity with that thread. Hence, UPC tries to minimize remote accesses as each thread and the part of the shared space that has affinity to it will likely be co-located in the same node. For a detailed description of all the language features, the reader can check [6].

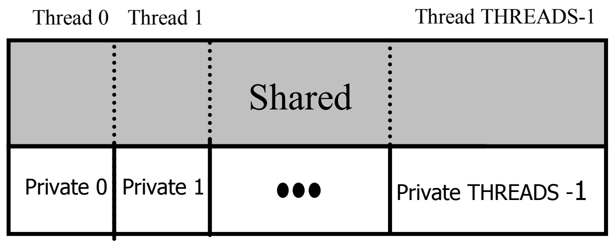

Figure 1. Distributed Shared Memory at UPC

The aim of this paper is to study the UPC features, from a programming point of view, that can make UPC a strong candidate for selection by programmers to develop high-performance applications productively.

In this paper we analyze the UPC language to see how it affects syntactic complexity, length, learning, and conceptual efforts. In order to do that, we provide some software productivity metrics and compare UPC to MPI for common standard high-performance computing benchmarking suites.

The rest of the paper is organized as follows. Section 2 examines UPC productivity, first by giving the necessary background about the related efforts done in software engineering on measuring productivity, followed by the main features that render UPC a productive language. Section 3 describes the methodology used in our study including metrics and benchmarks used in this paper. Section 4 presents and analyzes the comparative results for both UPC and MPI. Finally section 5 closes with concluding remarks.

2. UPC and Productivity

In this section, we give a brief background about the previous work done in measuring productivity and we discuss the main features that make UPC easy to use and hence increase the programmer productivity.

2.1 Programming Effort and Complexity

The groundwork of software measurement has been an open issue since the sixties. It has been know that program structure and modularity are important considerations for the development of reliable software [7]. We achieve higher reliability when software systems are highly modularized [8]. This, however, will not be considered here as it is a characteristic of the underlying sequential language paradigm.

The earliest software measure is the line of code (LOC), which is used till today [9]. The main idea behind LOC is that program length can be used as a predictor of program characteristics such as reliability and ease of maintenance. In addition we also consider the character of codes (COC) as the number of lines can also depend on the style of the programmer himself/herself and not necessarily the application needs, and because we believe that it is a useful measure for the manual effort exerted by the programmer.

One other famous software measurement that captures conceptual complexity is McCabe [10]. This measure is derived from graph theory. It computes the complexity of a program by computing the number of linearly independent paths in the program flow graph. A survey about the methods of software measurement can be found at [11]. A related conceptual complexity measure here will be the number of calls to specific MPI or UPC libraries and the number of keywords used that are specific to the used parallel paradigm used.

2.2 Main Features of UPC

UPC language has many features that make it easy to use for programmers. These features can be categorized as follows:

• **Simple Syntax**: A quick look at UPC specifications shows that UPC has a very simple syntax. For example, remote access can be done using simple C-like assignment, in the same way as local assignment. An expression like (a = b) can be local assignment if both a and b are at the thread affinity. However, if either or both variables are in remote memory, remote access will take place, still with the same simple assignment expression.

• **Very Simple Extension to C**: One of the main design goals of UPC is that it consists of very small set of extensions to the well know C-language. A dozen of keywords defined at max in UPC as an extension to C. New parallel features on the other hand attempted to maintain the C philosophy to keep the language familiar to C writers.

• **Not library Based**: This eliminates many function calls and argument passing and could be complex to make functions as general as possible and account for heterogeneous systems.

• **Application Consistent Domain Decomposition**: This facilitates greatly the conceptual effort. For example, with the same loop, and the computation part untouched, tasks can be distributed among threads. Mechanisms needed for domain decomposition, such as ghost zones which are used frequently in MPI are nearly irrelevant in UPC.

Figure 2 shows a simple example for a Sobel edge detection program, written in C then converted to UPC, and which demonstrates some of the aforementioned points. The left part of the figure includes the conventional sequential C, and the right part contains the UPC version. The changes needed to convert from C to UPC are shown in bold. At the first glance, we can see that even without prior knowledge of UPC, we can still understand the main structure of the program, due to the syntactical simplicity of UPC. Furthermore, we can see that the amount of UPC keywords used is not large, because UPC is a superset of C with small set of extensions only.

The shared declaration distributes the image data across the threads by chunks of rows. This is shared data and therefore, each chunk will have affinity to a given thread, but all image data can be accessed by all threads. Therefore, as the Sobel operator window crosses thread boundaries there will be no need to do any special management to guard zones. The outer “for loop” in the C program became a upc_forall loop. The upc_forall has the same exact first three fields as the C for. The fourth field is used to distribute independent iterations as tasks across the threads. The used syntax in this case makes each thread operates primarily on its local shared image data, but uses only remote data when the thread data boundaries are crosses.

```
#include <stdio.h>
#include <string.h>
#include <upc_relaxed.h>

#define BYTE unsigned char

/* image size */
#define N 128
/* global variables */
shared [N*N/THREADS] BYTE
 orig[N][N], edge[N][N];

int Sobel()
{
 int i, j, d1, d2;
 double magnitude;

 upc_forall( i=1; i<N-1; i++; &edge[i][0] )
 {
  for( j=1; j<N-1; j++ )
   {
    d1  = (int) orig[i-1][j+1]
            - orig[i-1][j-1];
    d1 += (int) (orig[i][j+1]
            - orig[i][j-1])<< 1;
    d1 += (int) orig[i+1][j+1]
            - orig[i+1][j-1];

    d2  = (int) orig[i-1][j-1]
            - orig[i+1][j-1];
    d2 += (int) (orig[i-1][j]
            - orig[i+1][j])<< 1;
    d2 += (int) orig[i-1][j+1]
            - orig[i+1][j+1];

    magnitude = sqrt( d1*d1 + d2*d2 );

    edge[i][j] = magnitude>255 ? 255 :
                          (BYTE) magnitude;
   }
 }
 return 0;
}
```

Figure 2. From C to UPC: Sobel Edge Detection Example

2.3 Conceptual Simplicity of UPC

To demonstrate the conceptual complexity of UPC, figure 3 shows how the benchmark GUPS (Giga Updates Per Second) is implemented in UPC as well as MPI. GUPS is a program that uses a random hashing function to update entries of a very large base table.

It can be clearly seen that UPC results in simpler implementation. That is, a fewer number of logical steps are needed. This means that the program can be less error-prone.

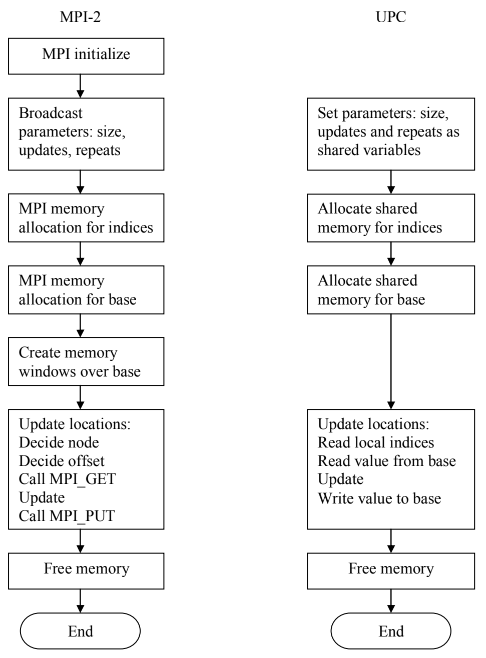
MPI-2
UPC
MPI initialize
Broadcast parameters: size, updates, repeats
MPI memory allocation for indices
MPI memory allocation for base
Create memory windows over base
Update locations: Decide node Decide offset Call MPI_GET Update Call MPI_PUT
Free memory
End
Set parameters: size, updates and repeats as shared variables
Allocate shared memory for indices
Allocate shared memory for base
Update locations: Read local indices Read value from base Update Write value to base
Free memory
End

Figure 3. MPI/2-GUPS VS UPC-GUPS

Furthermore, Figure 4 shows the memory view of both MPI-2 and UPS for GUPS. While the UPC code follows from the simple UPC memory model and translates into tables in which the table entries are directly accessed and manipulated, it was impossible to develop the application in MPI-1. As can be seen from the figure, the MPI-2 involves the creation of pseudo-shared windows, aligning the window over the local data structures and the transformation of the index from the local space to the pseudo-shared space. In addition, explicit put and get function calls to access the data are needed.

Domain Decomposition used:

upc_forall(i=0; i<UPDATES; i++; &indices[i])

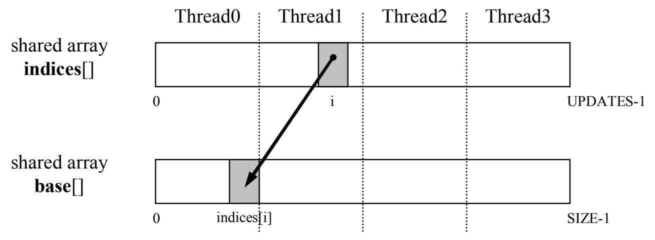

Figure 4.a UPC Memory View

Domain Decomposition used:

for(i=0; i<UPDATES/NP; i++)

with NP=#MPI Processes

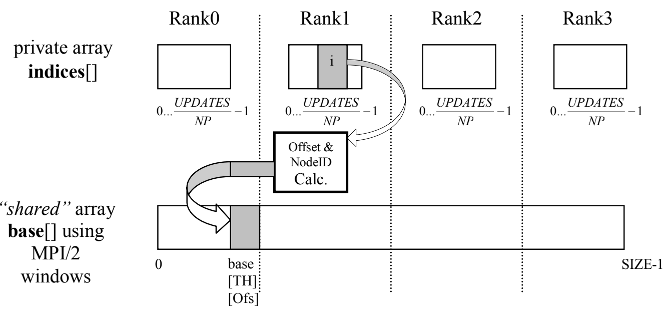

Figure 4.b MPI/2 Memory View

Figure 4. Memory View of MPI/2-GUPS VS UPC-GUPS

3. Methodology

This section we define the metrics that are used in our paper to measure the UPC productivity and the benchmarks used in this study.

3.1 Metrics Used

Our used metrics can be divided into two main categories. The first category is the manual programming effort. For that one of the metrics we have used is the lines of code or LOC [9]. In this metric we measure the lines of code used in the program, not including comments. We tried to have the optimum code for both MPI and UPC.

Although a lot of criticism has been given to this metric, it still gives valuable insight. This is because a small number of LOC means less effort required, and for the same programmer it means higher productivity for the same amount of time. LOC alone cannot give enough insights about the amount of effort the programmer does, because a single line can be very complicated in one language than in another language. This is why the second metric we have used is the number of characters (NOC). A small NOC coupled with small LOC means less manual effort.

The second category is the conceptual programming effort. This includes the effort done by the programmer to learn the language concepts, and apply them to go from the algorithmic representation of a problem to the source code. For a general way to capture the conceptual complexity, we count the number of parameters passed, as well as the number of function calls, types, etc, for different main tasks in that benchmark. We combine these counts to reach an overall score of complexity.

3.2 Benchmarks Used

In order to assess the productivity of UPC, we have compared it to MPI using the metrics mentioned above for a wide variety of applications. The applications used are described below.

Kernels of NAS Parallel Benchmark 2.0 [NPB]: [12] This benchmarks proposed by NASA Ames Research center tries to measure the performance of highly parallel computers. The following benchmarks are used from NPB suite.

• **CG (Conjugate Gradient)**: This benchmark computes an approximation to the smallest eigenvalue of symmetric positive definite matrix. Its main characteristic is an unstructured grid computation requiring irregular long-range communications.

• **EP (Embarrassingly Parallel)**: This benchmark requires little communication. It estimates the upper achievable limits for floating point performance of a parallel computer.

• **FT (Fast Fourier Transform)**: This benchmark solves a 3D partial differential equation using an FFT-based spectral method. It requires long range communication. Basically, it performs three one-dimensional (1-D) FFT s, one for each dimension.

• **IS (Integer Sorting)**: This benchmark is a parallel sorting program based on bucket sort. It requires a lot of total exchange communication.

• **MG (MultiGrid)**: The MG benchmark uses a V-cycle multigrid method to compute the solution of the 3-D scalar Poisson equation.

Other Kernels: To add variety to the applications used, we have added three more programs.

• **GUPS**: It is a program that uses a random hashing function to update entries of a very large base table.

• **Histogram**: It is a simple image histogramming program.

• **N-Queens**: In the N Queens problem we seek to find all solutions to the problem of placing N queens on an NxN chessboard such that no queen can kill the other. By chess rules, this means that no two queens may be placed on the same row, column, or diagonal. The method used is a depth-first searching and backtracking. For each row, we try to add a queen by checking the occupied columns and diagonals. The parallel solution to this problem is straightforward, due to the fact that the branches can be distributed across the threads with no thread interaction being needed.

4. Experimental Study

Tables 1 and 2 show the LOC for all the benchmarks for both UPC and MPI. Seq1 refers to sequential C code. Seq2 refers to the official NPB Serial code using the Fortran language. The sequential IS, GUPS, and Histogram are written only in the C language. The last two columns of the table show the effort in converting from the sequential code to the parallel code. It is calculated as: (LOC(UPC) - LOC(seq)) / (LOC(seq)).

The same equation is used for MPI effort. As it is shown in the table, it takes much less effort to parallelize a piece of code using UPC than MPI.

		SEQ	**UPC**	**UPC Effort (%)**
NPB-CG	#line	665	710	6.77
	#char	16145	17200	6.53
NPB-EP	#line	127	183	44.09
	#char	2868	4117	43.55
NPB-FT	#line	575	1018	77.04
	#char	13090	21672	65.56
NPB-IS	#line	353	528	49.58
	#char	7273	13114	80.31
NPB-MG	#line	610	866	41.97
	#char	14830	21990	48.28

		SEQ	**MPI**	**MPI Effort (%)**
NPB-CG	#line	506	1046	106.72
	#char	16485	37501	127.49
NPB-EP	#line	130	181	36.23
	#char	4741	6567	38.52
NPB-FT	#line	665	1278	92.18
	#char	22188	44348	99.87
NPB-IS	#line	353	627	77.62
	#char	7273	13324	83.20
NPB-MG	#line	885	1613	82.26
	#char	27129	50497	86.14

Table 1. Manual Efforts: NAS Kernels

As mentioned before, the LOC alone is not enough, because a single line may be very complicated. This is why the tables show the number of characters. Here UPC is consistently better than MPI when it comes to the effort of writing parallel code, in both LOC and NOCs, although in some benchmarks the difference is not high.

		SEQ	UPC	UPC Effort (%)
GUPS	#line	41	47	14.63
	#char	1063	1251	17.68
Histogram	#line	12	20	66.67
	#char	188	376	100.00
N-Queens	#line	86	139	61.63
	#char	1555	2516	61.80

		SEQ	MPI	MPI Effort (%)
GUPS	#line	41	98	139.02
	#char	1063	2979	180.02
Histogram	#line	12	30	150.00
	#char	188	705	275.00
N-Queens	#line	86	166	93.02
	#char	1555	3332	124.28

Table 2. Manual Efforts: Others Kernels

The second set of experiments has to do with the conceptual efforts. The results are shown in Table 3.

		Work Distr.	Data Distr.	Comm.	Synch. & Consist.	Misc. Ops	Sum	Overall Score
HISTOGRAM UPC	#Parameters	5	4	0	3	0	**12**	**22**
	#Function calls	0	0	0	4	0	**4**	
	#Keywords	2	1	0	0	0	**3**	
	#UPC Construct & UPC Types	0	2	0	1	0	**3**	
	Notes	2 if 1 for	2 shared decl.		1 lockdec 1 lock/unlock 2 barriers			
HISTOGRAM MPI	#Parameters	5	0	15	0	6	**26**	**47**
	#Function calls	0	0	2	2	4	**8**	
	# Keywords with rank & np	3	0	2	0	2	**5**	
	#MPI Types	0	0	6	0	2	**8**	
	Notes	2 if 1 for		1 Scatter 1 Reduce	(implicit w. Collective)	1 Init/Finalize 2 Comm		

Table 3.a Histogram

To show the conceptual complexity of language constructs in doing different tasks, we counted the number of keywords, function calls, parameters, for each of the common tasks needed in parallel programming. We then added them to get an overall score for MPI and UPC. Table 3 gives these statistics for both GUPS and Histogram. The rest of the benchmarks exhibits the same pattern. It should be noted here that as the application increases in size and complexity, the conceptual complexity of the language becomes more vital.

		Work Distr.	Data Distr.	Comm.	Synch. & Consist.	Misc. Ops	Sum	Overall Score
GUPS UPC	#Parameters	21	6	0	0	0	**27**	**43**
	#Function calls	0	4	0	2	0	**6**	
	#Keywords	3	4	0	0	0	**7**	
	#UPC Construct & UPC Types	3	0	0	0	0	**3**	
	Notes	3 forall 2 for 3 if	5 shared 2 all_alloc 2 free		2 barriers			
GUPS MPI	#Parameters	18	17	38	1	6	**80**	**136**
	#Function calls	0	7	6	3	6	**22**	
	# Keywords with rank & np	3	5	13	1	4	**26**	
	#MPI Types	0	6	2	0	0	**8**	
	Notes	5 for 3 if	2 mem alloc 2 mem free 3 window	2 one-sided 4 collect	(implicit w. Collective and WinFence) 1 barrier	Init Finalize comm_rank comm_size 2 Wtime (6 error handle)		

Table 3.b GUPS

Table 3. Statistics about Most, Common Languages Constructs

5. Conclusions

UPC is considered a strong candidate for being the main choice of parallel programming used in high-performance computing. In this paper we have discussed the important issue of UPC effect on programmer productivity. We stated the factors that affect the productivity of programming languages, namely the manual programming effort and conceptual or thinking effort. We then followed by presenting some metrics to measure the productivity of UPC. Although we understand that measuring programmer productivity is still an open issue in computer science, we believe that our metrics provided a much needed insight into the productivity of UPC as compared to MPI. While the question is how much exactly more productive is one paradigm over the other may remain debatable and it is application dependent, the important fact that is shown from our results is that UPC has shown consistent improvement over MPI in terms of lines of codes, number of characters, and conceptual effort.

References

[1] Melconian Terran, *Miscellaneous Comments on Programming Complexity* (http://www.consistent.org/terran/complex.shtml), October 2000

[2] El-Ghazawi Tarek, Carlson William and Draper Jesse, *UPC Language Specifications v1.1* (http://upc.gwu.edu), October 2003

[3] Culler, Dusseau Andrea, Goldstein Seth Copen, Krishnamurthy Arvind, Lumetta Steven, Von Eicken Thorsten and Yelick Katherine, *Parallel Programming in Split-C*, Proceedings of SuperComputing 1993, Portland, OR, November 15-19, 1993

[4] El-Ghazawi Tarek and Cantonnet François, *UPC Performance and Potential: A NPB Experimental Study*, SuperComputing 2002, IEEE, Baltimore MD, November 2002

[5] El-Ghazawi Tarek and Chauvin Sébastien, *UPC Benchmarking Issues*, 30th Annual Conference IEEE International Conference on Parallel Processing, 2001 (ICPP01) Pages: 365-372

[6] *Official UPC website*, GWU, 2004, http://upc.gwu.edu

[7] Zuse Horst, *A Framework of Software Measurement*, Published by Walter de Gruyter, 1998

[8] Schneidewind Norman F.. *Modularity Consideration in Real Time Operating Systems*. Computer Software and Applications Conference 1997 (COMPSAC), Pages 397-403

[9] Park Robert, *Software Size Measurement: A Framework for Counting Source Statements*, Technical report, Software Engineering Institute, 1992.

[10] McCabe Thomas, *A Complexity Measure*, IEEE Transactions of Software Engineering, SE-2(4), December 1976.

[11] Zuse Horst. *History of Software Measurement*. http://irb.cs.tu-berlin.de/~zuse/metrics/3-hist.html, September 1995

[12] *NAS Parallel Benchmark Suite*, NASA Advanced Supercomputing, 2002, http://www.nas.nasa.gov/Software/NPB