2011 IEEE 17th International Conference on Parallel and Distributed Systems

Design and Implementation of MapReduce using the PGAS Programming Model with UPC

Carlos Teijeiro, Guillermo L. Taboada, Juan Touriño, Ramón Doallo
Computer Architecture Group
Department of Electronics and Systems, University of A Coruña
Facultade de Informatica, Campus de Elviña s/n, 15071 A Coruña (Spain)

***Abstract*—MapReduce is a powerful tool for processing large data sets used by many applications running in distributed environments. However, despite the increasing number of computationally intensive problems that require low-latency communications, the adoption of MapReduce in High Performance Computing (HPC) is still emerging. Here languages based on the Partitioned Global Address Space (PGAS) programming model have shown to be a good choice for implementing parallel applications, in order to take advantage of the increasing number of cores per node and the programmability benefits achieved by their global memory view, such as the transparent access to remote data.**

This paper presents the first PGAS-based MapReduce implementation that uses the Unified Parallel C (UPC) language, which (1) obtains programmability benefits in parallel programming, (2) offers advanced configuration options to define a customized load distribution for different codes, and (3) overcomes performance penalties and bottlenecks that have traditionally prevented the deployment of MapReduce applications in HPC. The performance evaluation of representative applications on shared and distributed memory environments assesses the scalability of the presented MapReduce framework, confirming its suitability.

***Index Terms*—UPC, MapReduce, HPC, programmability, collective primitives**

I. Introduction

The implementation of coarse-grain parallelism to process large data sets in a distributed environment has become a very relevant issue nowadays. Here the use of the MapReduce [1] paradigm has proved to be a suitable solution in a wide variety of environments and applications. It consists of two phases: the “Map” stage, in which a function is applied in parallel to a set of input elements to generate another set of intermediate key/value pairs, and the “Reduce” stage, in which all the intermediate pairs with the same key are combined to obtain the final output. MapReduce is being used in many application fields, such as information retrieval [2], pattern recognition [3] or processing of medical information [4].

Regarding current HPC platforms, the use of multicore clusters with heterogeneous processors and hardware accelerators, such as GPUs and FPGAs, is increasing because of their capabilities for data processing. However, the use of MapReduce for HPC requires high efficiency, and the exploitation of data locality with parallel languages and libraries. Therefore, a good approach is the use of new programming paradigms to provide efficiency and programmability in code development, such as PGAS [5]. The PGAS model is an emerging paradigm in which the memory is viewed as a global address space divided in two areas, private and shared, both of them logically partitioned in chunks that have affinity to different threads. Thus, each thread has its own shared memory area that is used to process data locally. The most relevant PGAS languages are extensions of popular languages, such as C (UPC), Java (Titanium) and Fortran (Co-array Fortran).

This work presents an implementation of MapReduce using the PGAS programming model. The goal of this new approach is to put together the potential of MapReduce for large data sets and the programmability and efficiency of PGAS languages in order to manage current and upcoming hardware architectures. The UPC language [6] has been selected to implement this framework, because it merges the powerful features of C with constructs for parallel programming, such as collective functions and remote memory copies using assignments. The proposed MapReduce implementation has been designed to be generic, that is, with ability to accept any type of `map` and `reduce` operations defined by the programmer. In order to illustrate the behavior of this framework, three different applications have been selected for their analysis on shared and distributed memory environments.

The rest of this paper is structured in the following sections. Section 2 presents the state of the art in MapReduce implementations. Section 3 discusses the design and implementation of MapReduce in UPC, explaining the most relevant design decisions taken. Section 4 presents performance results using three representative applications. Finally, Section 5 summarizes the main conclusions of this work.

II. Related Work

Currently, following the guidelines of the first MapReduce framework published by Google [7], many other implementations of this framework using different languages have been developed. In general, most of the existing MapReduce implementations are written using object-oriented languages: for example, the Google MapReduce framework was developed in C++, and a popular open source implementation of MapReduce included in the Apache Hadoop project [8] was written in Java.

The use of MapReduce has been generally applied to large data-parallel problems that deal with the processing of large sets of files, and these kinds of problems have also been used as testbed for performance evaluations [9]. As these tasks are

1521-9097/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPADS.2011.162

IEEE computer society

```
int map(void *input, void *key, void *value);
int reduce(void *key, void *value, int nelems, void *result);

int ApplyMap(
    int (*mapfunc)(void *,void *,void *), void *inputElems, int nelems,
    int userDefDistrFlag, int algorithm, int *weights);
void *ApplyReduce(
    int (*reducefunc)(void *,void *,int,void *), int nelems, int gathFlag,
    int collFlag, int sizeKey, int sizeValue);
```

Listing 1. Signatures of the basic functions in UPC MapReduce

usually integrated in applications written in Java or C++, all these works have been focused on those languages using distributed memory environments. There is also a recent work on the implementation of MapReduce for the X10 programming language [10], which uses PGAS as basis for parallelism, focusing specifically on exploiting programmability. The use of X10 provides a Java-like approach that benefits of dealing with heterogeneous platforms using a tuned virtual machine. Even though the use of high-level structures is interesting for general-purpose distributed applications, this type of processing does not fit naturally on many HPC applications, such as the simulation of biological processes, where performance plays the most important role.

Although there is still very little work on MapReduce applied to HPC, some interesting frameworks have been implemented in C on shared memory, such as the Phoenix project [11] and Metis [12], or using MPI C (linked to a C++ library) for distributed memory environments, like MapReduce-MPI [13]. Additionally, a study of the possibilities of implementing an optimized version of MapReduce with MPI [14] concluded that it is feasible, but additional features to favor productivity and performance in MPI, such as improved collective communications, are needed. Therefore, our framework has been developed in order to overcome these previous limitations in MapReduce support for HPC and provide a simple and efficient implementation for both shared and distributed memory environments based on UPC and PGAS.

III. Design and Implementation of UPC MapReduce

The UPC MapReduce code is structured in two types of functions: the generic management functions and the user-defined functions. Management functions are used to support the UPC MapReduce functionality, that is, they initialize the framework and perform the necessary communications between threads for work distribution at "Map" and data collection at "Reduce". User-defined functions include the processing that should be performed on each element at the map stage and on the intermediate elements at the reduction stage. In order to maximize simplicity and usability, some basic ideas have guided the implementation of the UPC MapReduce framework:

- No parallel code development is needed: the management functions can perform all the parallel work, thus the user can simply write the sequential C code for the `map` and `reduce` functions that will be applied to the input and intermediate elements, respectively. However, if more control on the work distribution at the "Map" stage is desired, the framework also allows the user to disable all these mechanisms and define a custom parallelization routine within the `map` function.
- Simplicity in management: the generic management functions for MapReduce should also be written in an expressive and simple way. If the user needs to modify them, the use of a clear coding favors a better understanding of the processing of the management functions. Traditionally, UPC code has always tried to exploit performance by using privatizations and direct data movements using bulk memory copies [15], but here the UPC MapReduce management codes tend to use higher level constructs, such as collective functions, that encapsulate data movements between threads.
- Reusability and flexibility: the tasks implemented by the management functions are kept as generic as possible. Thus, the parameters to these functions are treated as a homogeneous set of values, in which each value has to be processed in the same way, regardless of its type. This generic typing is obtained by means of arrays of void type (void *), and the correct interpretation of input and output data to these functions is left to the user, because it can vary depending on the application in which MapReduce is used.

The piece of code included in Listing 1 presents the signatures of the two user-defined `map` and `reduce` functions in this framework, and the MapReduce management functions that perform the mapping (`ApplyMap`) and the reduction (`ApplyReduce`). The next subsections give a more detailed description of the implementation process of the latter two, and also some general remarks on the UPC implementation of the framework.

A. Function `ApplyMap`

As can be seen in Listing 1, this function receives six input parameters: (1) the function that should be used for the "Map" stage, (2) the list of elements to which it has to be applied, (3) the number of elements in that list, (4) a flag (`userDefDistrFlag`) that indicates whether the work distribution is performed by `ApplyMap` or the user

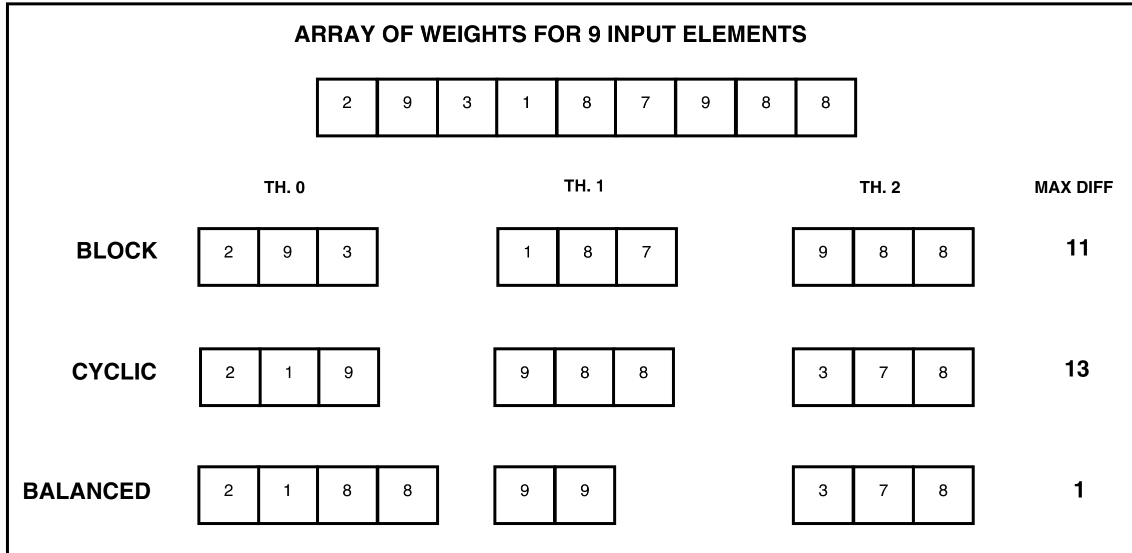


Figure 1. Examples of BLOCK, CYCLIC and BALANCED work distribution

defines a custom distribution, (5) an integer that identifies the distribution algorithm, and (6) an array of weights that indicates the workload assigned to each input element in the list. According to this interface, first each thread must have the whole list of input elements stored in their private memory before calling to `ApplyMap`, and then the list is split in subsets of elements that should be processed by each thread. The work distribution flag determines the type of processing used in `ApplyMap`. If this flag is not enabled, `ApplyMap` distributes the input elements between threads according to the algorithm selector passed as parameter, and the `map` function provided by the user is applied to each single element to generate an intermediate key/value pair (from now on, this will be referred as "element-by-element map"). If the flag is enabled, the following two parameters in `ApplyMap` are ignored, and the work distribution routine is performed inside the `map` function for the whole set of input elements ("array-based map"), thus returning a set of intermediate elements. As a result of this, the implementation of the `map` function (defined by the user) must be consistent with the work distribution selected in `ApplyMap`. When an element-by-element processing is chosen, `ApplyMap` implements four different algorithms, that are selected using the following values for the `algorithm` parameter:

- `BLOCK`: this algorithm divides the number of input elements according to a block distribution.
- `CYCLIC`: this option assigns each input element to a thread in a cyclic (round-robin) way, according to the element position in the input array and the thread identifier. The block and cyclic algorithms do not require the definition of the `weights` parameter.
- `BALANCED`: this algorithm implements a heuristic to obtain a load-balanced distribution of input elements between threads according to the values passed as parameters in the `weights` array (positive integer values). The basis is that a high value for a weight indicates that the associated input element presents a high workload, thus each element in the input vector is assigned to the thread whose current associated workload (according to the value defined in a counter) presents the lowest value of all threads.
- `BEST_FIT`: this option indicates the selection of the most efficient algorithm among the three previous candidates. To do this in a compact way, the first element in the array of weights represents here a threshold value, and the weights associated to the input elements are stored in the following array positions. The threshold value indicates the maximum difference between workloads for different threads that may be considered as acceptable in order to have a load-balanced execution. The workloads for the block and cyclic algorithms are computed, and if the highest workload difference is below the threshold for one of them, that algorithm is selected for work distribution (if both pass the test, the block distribution is selected by default). If none of these is suitable, the balanced algorithm is selected.

Figure 1 presents an example of work distribution among three threads using a set of 9 input elements with different computational weights, with the maximum difference of workload between threads. Here the BALANCED algorithm obtains a more similar distribution of workload between threads than the BLOCK or CYCLIC ones, because it takes advantage of the implemented heuristic. Consequently, for this input set the BEST_FIT algorithm would select the BALANCED distribution.

It is important to remark that after the execution of `ApplyMap` each thread keeps its own sublist of intermediate elements stored in the private memory space, which is

completely independent from the rest of the threads. These intermediate elements are managed by the MapReduce framework transparently to the user, thus they do not need to be returned by `ApplyMap`.

B. Function `ApplyReduce`

This function takes six parameters (see Listing 1): (1) the function that will be used to combine the elements produced by each thread in the "Map" stage, (2) the number of intermediate key/value pairs per thread, (3) the gathering flag, (4) the collective communications flag (`collFlag`), (5) the size of each intermediate key, and (6) the size of each intermediate value. The gathering flag (`gathFlag`) plays an important role in deciding the necessary communications between threads at the beginning of `ApplyReduce`. If its value is `NOCOMM`, it indicates that each thread only requires its intermediate values to return the final result, thus no communications are performed. When set to `ALLCOMM`, all threads should gather the intermediate data from the rest of the threads in order to compute the reduction. Otherwise, a positive integer value from 0 to `THREADS-1` (being `THREADS` the number of threads in the UPC execution) indicates the identifier of the only thread that should gather the intermediate data and compute the final result.

The communications required here by `ApplyReduce` are performed at the same time by all threads, so collective functions can be used. However, UPC MapReduce cannot apply here directly the standard UPC collective primitives, for two major reasons: (1) MapReduce operates on each thread's private memory space, but standard UPC collectives must use shared arrays as source and destination, and (2) some collective communications, such as the gathering of intermediate elements, generally have a different number of elements per thread, which is not supported with the standard UPC collectives. Thus, in order to solve these issues and improve programmability, a set of extended UPC collective operations has been implemented. They are based on different proposals by the UPC community for extending the standard collectives [16], including the use of private arrays as source and destination, and communications with a different number of elements in each thread (vector-variant collectives). The decision of implementing a set of collectives is justified because the use of these collectives is not restricted to `ApplyReduce`, as they are also useful for some implementations of the `map` function for array-based map. These collectives use efficient algorithms for multicore cluster architectures, implementing binomial-tree communications between nodes and flat-tree communications on shared memory within a node, and are completely written in UPC using the standard memory copy primitives, therefore they are portable. Their use in `ApplyReduce` for communications between threads is controlled using the collective communications flag (`collFlag` parameter in Listing 1). If this flag is enabled, collective functions are used for communication, and the user should indicate the size of each key and value in the set of intermediate elements as parameters to `ApplyReduce`. When `collFlag` is not enabled, the intermediate elements are transferred one by one to their corresponding destination(s). The only requirement to use collectives in `ApplyReduce` is that all intermediate keys and values must have the same size, but this situation can be considered as the most common case in practice. Listing 2 presents the signature of the extended allgather collective used in `ApplyReduce` compared to the standard UPC allgather: the vector-variant collective (named `upc_all_gather_all_v_priv`) can read any amount of data (`nelems`) with any given displacement in the source and destination arrays (`sdisp` and `ddisp`, respectively), and also being aware of the size of each element (`typesize`).

After the data communication step in `ApplyReduce` (if required), the reduce function (first parameter) is applied to the set of intermediate elements stored in the private memory space of each thread. This user-defined function receives as input parameters two arrays of keys and values (that represent the set of intermediate elements) and the number of elements. The combination of all these intermediate elements is considered as the final result of MapReduce, and it is returned by one or all threads. If no communications between threads took place in the previous step, each thread returns a portion of the final result, and if communications were performed, all threads (or only the selected thread) return the complete final result.

The implemented approach establishes that the `reduce` function defined by the user must always process all the intermediate elements associated to each thread in a single call, analogously to the definition of the array-based map in `ApplyMap` commented in Section III-A. This design decision allows a more flexible definition of customized reduction functions for the user. An alternative design for `ApplyReduce` could have also allowed the use of an element-by-element reduction, but this type of processing would only be useful for a restricted set of reduction operations on basic datatypes (e.g. a sum of integers or a logical AND operation), and it would be difficult to perform many other operations. For example, the computation of the average value in a list of integers would imply either the definition of a variable number of parameters for the element-by-element reduce operation (which would complicate the design of the function and thus cause unnecessary trouble to the user), or the transfer of the essential part of the processing to the `ApplyReduce` function (which would not be acceptable for abstraction purposes). Therefore, the definition of the reduction on the complete set of intermediate elements has been considered as the best choice for this implementation.

IV. Performance Evaluation

This section includes performance results for UPC MapReduce on shared and distributed memory compared to the Phoenix system [17] on shared memory, and the MapReduce-MPI framework [13] for distributed memory. Both frameworks rely on C++ libraries, but the MapReduce codes developed with them can be written in C (using the MPI library for MapReduce-MPI), therefore they can offer better performance

```
void upc_all_gather_all (
    shared void *dst, shared const void *src, size_t nbytes, upc_flag_t mode)

void upc_all_gather_all_v_priv (
    void *dst, const void *src, shared int *ddisp, shared int *sdisp,
    shared int *nelems, size_t src_blk, size_t typesize, upc_flag_t mode)
```

Listing 2. Signatures of the standard UPC allgather collective and the extended vector-variant private collective used in `ApplyReduce`

than other approaches that use Java codes, as stated in previous evaluations [9].

A. Experimental Conditions

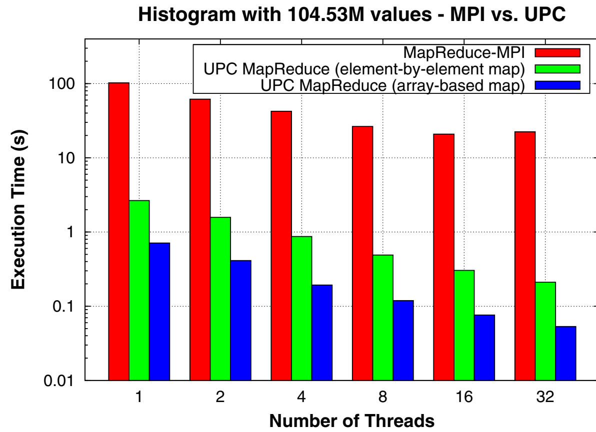
In order to evaluate the MapReduce framework, three applications have been selected: (1) Histogram (*Hist*), (2) Linear Regression (*LinReg*) and (3) Word Count (*WCount*). *Hist* obtains the values of the pixels in a bitmap image (values for RGB colors, ranging from 0 to 255) and computes the histogram of the image, that is, it counts the number of pixels that have the same value for a color. *LinReg* takes two lists of double-precision paired values that represent coordinates, and computes the line that fits best for them. *WCount* processes a set of input text files in order to count the number of occurrences of distinct words. The UPC codes for *Hist* and *LinReg* are based on the ones included in the Phoenix distribution, although some changes have been made in order to perform representative tests (e.g. the input values to *LinReg* used here are double-precision numbers) and provide analogous codes for UPC, C and MPI. It is important to note that, according to the approaches described in Section III-A, Phoenix always performs an array-based map at the "Map" stage, whereas MapReduce-MPI always uses an element-by-element map; thus, the UPC code has been adapted to have a fair comparison with both approaches. Additionally, the work distribution for all tests has been implemented using a block algorithm, and collectives are used for all codes except *WCount*. The input images for *Hist* were obtained from the Phoenix web page, the *LinReg* coordinates were randomly generated and the files for *WCount* were taken from the SPAM corpus in the TREC Conference collection [18], widely used in information retrieval.

The UPC MapReduce implementation has been evaluated using two different testbeds. The first one (named SMP) is a multicore system with 2 Intel Xeon E5520 processors (4 cores per processor with hyper-threading enabled, thus 8 cores per node which are able to run 16 threads simultaneously), and 8 GB of memory. The second one (FT) is the Finis Terrae supercomputer installed at the Galicia Supercomputing Center (CESGA) [19], which has 142 nodes with 16 cores per node in 8 MontVale Itanium2 (IA64) dual-core processors at 1.6 GHz, 128 GB of memory per node and InfiniBand network. In both environments, the UPC compiler used was Berkeley UPC [20] (BUPC) 2.12.1 (released in December 2010), with Intel icc 11.1 as C background compiler. BUPC uses threads for intra-node communication and InfiniBand Verbs transfers for inter-node communication. The HP MPI 3.3-005 library was used in FT by MapReduce-MPI.

The SMP environment has been used for shared memory executions, whereas the FT is only able to perform distributed memory executions. The reason is that the optimizations in Phoenix do not support the IA64 architecture of FT. Only the most representative results for the three applications on shared and/or distributed memory have been included because of space limitations. Regarding executions on distributed memory, all executions maximize the number of threads (or MPI processes) per node: all tests with up to 16 threads in FT were executed on the same node, 2 nodes were used for 32 threads, and so forth.

B. Performance Results

Figure 2 presents the most relevant comparative execution times on distributed memory (for InfiniBand communications) using UPC and MPI for *Hist* and *WCount*. The graph on the left shows that the execution times of UPC are clearly lower than those of MPI for *Hist* using 104.53 millions of input values (only the results up to 32 threads are shown because for a larger number of threads MapReduce-MPI does not scale), but in the graph on the right the *WCount* execution times with 75419 input files are very similar, achieving a speedup of about 75 for 128 threads with both implementations. The reasons for this behavior are the amount of input elements used in each code and the implementation of MapReduce support on MPI using C++ libraries. Even though the computational load for *WCount* is clearly higher than for *Hist* in these experiments, UPC MapReduce presents a more lightweight implementation than MapReduce-MPI. We have observed that 90% of the sequential execution time of MapReduce-MPI for *Hist* is spent in handling the C++ library structures associated to every input element, whereas the same percentage falls to an 8% for the presented *WCount* problem size (values obtained from a performance profiling). Therefore, when there is a very large number of input elements with little computation (value testing and clustering in *Hist*), the element-by-element map processing performed by MapReduce-MPI involves much more overhead than processing a reduced number of input elements with more intensive computation for each one (obtaining the words in a file in *WCount*). Nevertheless, the element-by-element map in UPC MapReduce is not affected by this circumstance because of its pure UPC implementation, thus it obtains better results for *Hist*, and very similar performance for *WCount*. As a consequence of this, MPI results are not shown in the following figures for *LinReg* and *Hist*, because of their poor performance with a large number of input elements.



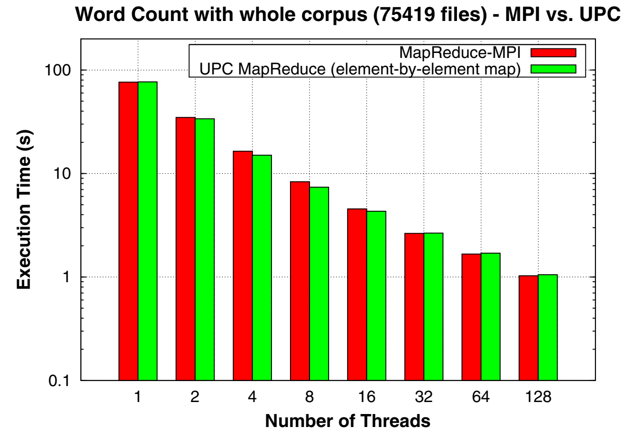
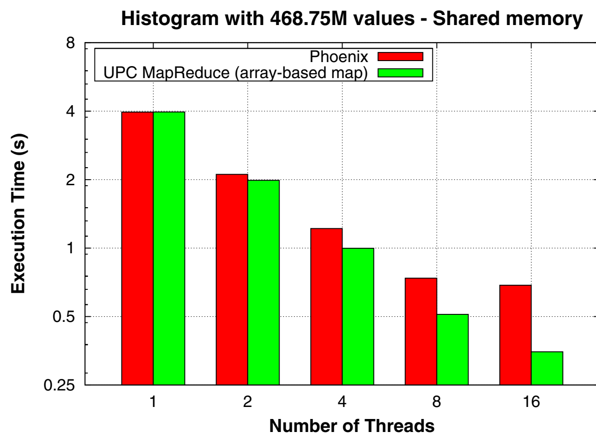


Figure 2. Performance comparison of MapReduce-MPI and UPC MapReduce on distributed memory (FT)



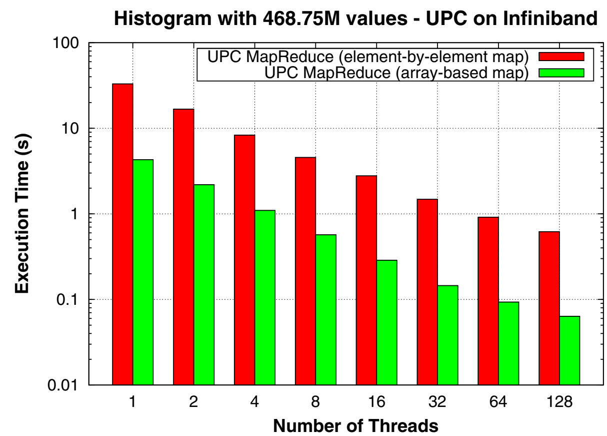
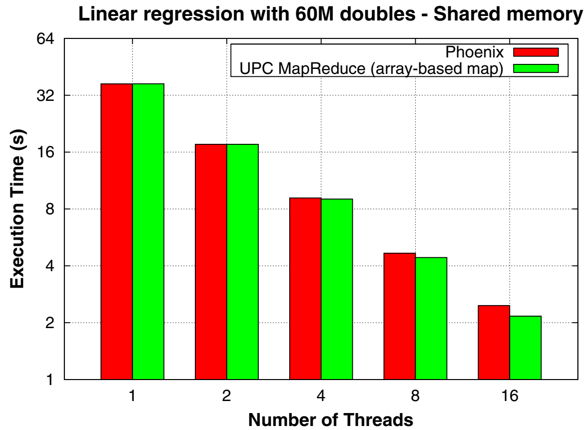


Figure 3. Performance of *Hist* on shared and distributed memory (SMP and FT, respectively)

Additionally, the graph on the left shows that the use of an array-based map with UPC for *Hist* obtains better results than the default element-by-element map in `ApplyMap`, because the array-based approach allows the user to define an optimized `map` implementation for the whole set of input data, and not just for every single element separately. Regarding the *WCount* code, these differences are negligible, therefore the results obtained using an array-based map are not shown in the graph on the right. The possibility of implementing an optimized array-based map for a given code greatly depends on the nature of input data. For instance, the optimization performed here for the *Hist* code consists in reading all input data as a single array of characters, distributing it in `THREADS` balanced chunks (one per thread) and classifying the elements using bytewise comparison. This procedure can be implemented because all input elements in *Hist* have the same size (`sizeof(int)`), but it is impossible to do the same optimization for *WCount* because the words have different length. As a result of this, when the use of an optimized array-based map is possible, it is the best option to obtain the best performance. However, for a low or medium amount of input data, such as in *WCount*, the programmers can safely rely on the default element-by-element map, which does not require the implementation of UPC routines for data distribution.

Figure 3 shows performance results for *Hist* using 468.75 millions of input values. In our SMP testbed, the UPC implementation of *Hist* has better performance on shared memory than Phoenix. This is related to the amount of input elements processed and the use of shared C++ objects in Phoenix to process the intermediate data, which involve a higher overhead than the UPC processing. Additionally, one main reason for this behavior is that the current version of the BUPC compiler includes optimized shared memory communications through threads, obtaining better performance. Unlike MapReduce-MPI, Phoenix has similar execution times than UPC MapReduce, mainly because of the use of an array-based approach at the "Map" stage. Regarding distributed memory (InfiniBand communications), the graph on the right presents the execution times for the UPC implementation in order to show that



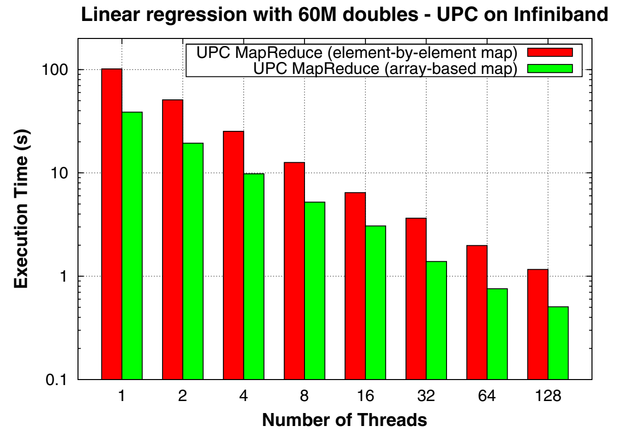


Figure 4. Performance of *LinReg* on shared and distributed memory (SMP and FT, respectively)

the use of an array-based map is very relevant to obtain lower execution times, because of the use of the optimizations commented for the graph on the left in Figure 2.

Figure 4 presents the results of the *LinReg* code using 60 millions of input values. These graphs confirm that UPC presents slightly better results than Phoenix for *LinReg* on shared memory. Although the global computational workload of this test is similar to that of *Hist* shown in Figure 3, here the differences are smaller because the amount of input elements is also smaller and thus the Phoenix C++ library overhead is reduced. We have checked that the execution times for *WCount* on shared memory (not shown for simplicity) follow the same trend: they are almost similar for Phoenix and UPC (tested with the whole TREC corpus, 75419 files). Once again, the graph on the right of Figure 4 shows that the use of an array-based map is a key factor to obtain better performance.

V. Conclusions

This paper has presented the first (to our knowledge) implementation of MapReduce using UPC that is intended to take advantage of the productive programming of the PGAS memory model: it provides a shared memory view of distributed memory systems, thus increasing programmability while allowing data locality exploitation. This UPC MapReduce implementation consists of two management functions for each stage of processing (`ApplyMap` and `ApplyReduce`) that can manipulate any kind of input elements in a generic way, using two sequential user-defined `map` and `reduce` functions and allowing flexible control on the workload distribution at the "Map" stage, either automatically or user-defined. This framework has been designed and implemented on a basis of simplicity and usability, exploiting language facilities such as collective functions, but always focusing on obtaining good performance.

The UPC implementation of MapReduce has been evaluated on shared and distributed memory environments using three representative applications. The UPC MapReduce codes have been compared to alternative frameworks (Phoenix and MapReduce-MPI). According to the evaluations accomplished, UPC MapReduce has obtained similar or better results than those approaches on all shared and distributed memory executions, because of the use of a generic and simple UPC framework that is based only on its own language constructs, without calling external libraries, and also relying on efficient communications at low level, specially on shared memory. As a result, UPC has proved to be a good and feasible solution for implementing a MapReduce framework in order to execute codes on different HPC environments, offering programmability without performance penalties, even being able to achieve performance advantages.

Acknowledgments

This work was funded by Hewlett-Packard and partially supported by the Spanish Ministry of Science and Innovation under Project TIN2010-16735. We gratefully thank CESGA for providing access to the Finis Terrae supercomputer.

References

[1] J. Dean and S. Ghemawat, "MapReduce: A Flexible Data Processing Tool," *Communications of the ACM*, vol. 53, no. 1, pp. pp. 72–77, 2010.

[2] R. M. C. McCreadie, C. Macdonald, and I. Ounis, "On Single-Pass Indexing with MapReduce," in *32nd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'09), Boston (MA, USA)*, 2009, pp. 742–743.

[3] P. Krajca and V. Vychodil, "Distributed Algorithm for Computing Formal Concepts Using Map-Reduce Framework," in *8th International Symposium on Intelligent Data Analysis (IDA'09), Lyon (France)*, 2009, pp. 333–344.

[4] S. J. Matthews and T. L. Williams, "MrsRF: an Efficient MapReduce Algorithm for Analyzing Large Collections of Evolutionary Trees," *BMC Bioinformatics 2010*, vol. 11, no. Suppl. 1:S15, 2010.

[5] W. Carlson, T. El-Ghazawi, R. Numrich, and K. Yelick, "Programming in the Partitioned Global Address Space Model," in *Tutorial at the 15th Conference on High Performance Networking and Computing (SC'03), Phoenix (AZ, USA)*, 2003.

[6] Unified Parallel C at George Washington University. http://upc.gwu.edu [Last visited: May 2011].

[7] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *6th Symposium on Operating System Design and Implementation (OSDI'04), San Francisco (CA, USA)*, 2004, pp. 137–150.

[8] Hadoop Project. http://hadoop.apache.org [Last visited: May 2011].

[9] C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," in *13th International Conference on High-Performance Computer Architecture (HPCA'07), Phoenix (AZ, USA)*, 2007, pp. 13–24.

[10] H. Dong, S. Zhou, and D. Grove, "X10-Enabled MapReduce," in *4th Conference on Partitioned Global Address Space Programming Model (PGAS'10), New York (NY, USA)*, 2010.

[11] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System," in *2009 IEEE International Symposium on Workload Characterization (IISWC'09), Austin (TX, USA)*, 2009, pp. 198–207.

[12] Y. Mao, R. Morris, and F. Kaashoek, "Optimizing MapReduce for Multicore Architectures," in *Technical Report MIT-CSAIL-TR-2010-020, MIT*, 2010.

[13] MapReduce-MPI Library. http://www.sandia.gov/˜sjplimp/mapreduce.html [Last visited: May 2011].

[14] T. Hoefler, A. Lumsdaine, and J. Dongarra, "Towards Efficient MapReduce Using MPI," in *16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'09), Espoo (Finland)*, 2009, pp. 240–249.

[15] T. El-Ghazawi and S. Chauvin, "UPC Benchmarking Issues," in *30th International Conference on Parallel Processing (ICPP'02), Valencia (Spain)*, 2001, pp. 365–372.

[16] Z. Ryne and S. Seidel. UPC Extended Collective Operations Specification. http://www.upc.mtu.edu/papers/UPC_CollExt.pdf [Last visited: May 2011].

[17] The Phoenix System for MapReduce Programming. http://mapreduce.stanford.edu [Last visited: May 2011].

[18] Text Retrieval Conference (TREC). http://trec.nist.gov [Last visited: May 2011].

[19] Galicia Supercomputing Center (CESGA). http://www.cesga.es/index.php?lang=en [Last visited: May 2011].

[20] Berkeley UPC Project. http://upc.lbl.gov [Last visited: May 2011].