Victor Malyshkin (Ed.)

LNCS 10421

Parallel Computing Technologies

14th International Conference, PaCT 2017
Nizhny Novgorod, Russia, September 4–8, 2017
Proceedings

Springer

Lecture Notes in Computer Science 10421

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Victor Malyshkin (Ed.)

Parallel Computing Technologies

14th International Conference, PaCT 2017
Nizhny Novgorod, Russia, September 4–8, 2017
Proceedings

Springer

Editor
Victor Malyshkin
Russian Academy of Sciences
Novosibirsk
Russia

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-62931-5 ISBN 978-3-319-62932-2 (eBook)
DOI 10.1007/978-3-319-62932-2

Library of Congress Control Number: 2017946060

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

© Springer International Publishing AG 2017
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

A New Class of the Smallest Four-State Partial FSSP Solutions for One-Dimensional Ring Cellular Automata 232
Hiroshi Umeo and Naoki Kamikawa

Properties of the Conservative Parallel Discrete Event Simulation Algorithm 246
Liliia Ziganurova and Lev Shchur

Organization of Parallel Computation

Combining Parallelization with Overlaps and Optimization of Cache Memory Usage 257
S.G. Ammaev, L.R. Gervich, and B.Y. Steinberg

Defining Order of Execution in Aspect Programming Language 265
Sergey Arykov

Automated GPU Support in LuNA Fragmented Programming System 272
Belyaev Nikolay and Vladislav Perepelkin

Automation Development Framework of Scalable Scientific Web Applications Based on Subject Domain Knowledge 278
Igor V. Bychkov, Gennady A. Oparin, Vera G. Bogdanova, Anton A. Pashinin, and Sergey A. Gorsky

Stopwatch Automata-Based Model for Efficient Schedulability Analysis of Modular Computer Systems 289
Alevtina Glonina and Anatoly Bahmurov

Parallelizing Inline Data Reduction Operations for Primary Storage Systems 301
Jeonghyeon Ma and Chanik Park

Distributed Algorithm of Dynamic Multidimensional Data Mapping on Multidimensional Multicomputer in the LuNA Fragmented Programming System 308
Victor E. Malyshkin and Georgy A. Schukin

Probabilistic Causal Message Ordering 315
Achour Mostéfaoui and Stéphane Weiss

An Experimental Study of Workflow Scheduling Algorithms for Heterogeneous Systems 327
Alexey Nazarenko and Oleg Sukhoroslov

PGAS Approach to Implement Mapreduce Framework Based on UPC Language 342
Shomanov Aday, Akhmed-Zaki Darkhan, and Mansurova Madina

PGAS Approach to Implement Mapreduce Framework Based on UPC Language

Shomanov Aday(✉), Akhmed-Zaki Darkhan, and Mansurova Madina

Al-Farabi Kazakh National University, Almaty, Kazakhstan
`adai.shomanov@gmail.com, {darhan_a,mansurova01}@mail.ru`

Abstract. Over the years from its introduction Mapreduce technology proved to be very effective parallel programming technique to process large volumes of data. One of the most prevalent implementations of Mapreduce is Hadoop framework and Google proprietary Mapreduce system.

Out of other notable implementations one should mention recent PGAS (partitioned global address space) – based X10, UPC (Unified Parallel C) versions. These implementations present a new viewpoint when Mapreduce application developers can benefit from using global address space model while writing data parallel tasks. In this paper we introduce a novel UPC implementation of Mapreduce technology based on idea of using purely UPC based implementation of shared hashmap data structure as an intermediate key/value store. Shared hashmap is used in to perform exchange of key/values between parallel UPC threads during shuffle phase of Mapreduce framework. The framework also allows to express data parallel applications using simple sequential code.

Additionally, we present a heuristic approach based on genetic algorithm that could efficiently perform load balancing optimization to distribute key/values among threads such that we minimize data movement operations and evenly distribute computational workload.

Results of evaluation of Mapreduce on UPC framework based on WordCount benchmark application are presented and compared to Apache Hadoop implementation.

Keywords: UPC · PGAS · Mapreduce

1 Introduction

Large-scale data processing nowadays is widely used in many domains of science and industry. There is a large number of sophisticated tools and algorithmic solutions that allow to achieve high efficiency in handling and processing enormous amount of data. Main driving forces of modern big data development are powerful Mapreduce - based frameworks. The idea of Mapreduce was first presented in paper [1] by Google researchers Jeffrey Dean and Sanjay Ghemawat in 2004. In general, the main idea behind Mapreduce is to divide processing of the big data set between concurrently running map and reduce processes such that each process performs processing of smaller data chunk. The processing work in Mapreduce is done in several steps:

V. Malyshkin (Ed.): PaCT 2017, LNCS 10421, pp. 342–350, 2017.
DOI: 10.1007/978-3-319-62932-2_33

- **Init phase.** Specify map and reduce functions, provide input and output directory paths and etc.
- **Map phase.** Each mapper scans the input chunk of data and emits key/value pairs based on user provided map function.
- **Shuffle phase**. Distribute key/value pairs among reducers in a way that each reducer operates on list of key/value pairs with some assigned to that reducer unique key.
- **Reduce phase.** Each reducer performs operations on assigned key based on user provided reduce function.

The main complexity in efficiently implementing Mapreduce lies in developing scalable and optimized code for shuffle phase. To achieve these goals it is required to distribute key/value pairs with minimized network latencies. In distributed environment due to necessity of data movements between processes that belong to different nodes, network latencies can be very high and significantly degrade overall performance.

To overcome that we need to consider efficient tools that will allow to perform sufficiently transparent and optimized remote data access operations. For that purpose in our current work we will use UPC programming language. UPC programming language [2] belongs to a family of PGAS languages. PGAS (Partitioned Global Address Space) is a parallel programming model in which memory address space is divided into two non-overlapping logical areas: private and shared. Private space is local to every thread and can be accessed only by its own thread. Shared space has a more complex structure where each thread has an access to shared memory and each memory element has additionally affinity to the owner thread. The benefit of PGAS model is that each thread has a transparent view of shared memory layout hence locality can be preserved where it is needed to optimize data distribution for specific purposes of the application. UPC language provides a set of operations with shared memory such as: pointers arithmetic, write and read functions, memory allocation and de-allocation functions and other. UPC uses specifically designed GasNet communication system that enables high-performance one-sided communications in order to implement remote data access operations on shared memory.

2 Related Work

The implementation of PGAS-based Mapreduce model requires careful consideration and solving of many problems associated with the organization of the computational process, the process of data exchange between computing nodes, distribution and load balancing between concurrent map and reduce processes. In the article [3], authors describe Mapreduce framework, implemented on the UPC language. The approach described in this article applies collective functions for data exchange in shuffle phase. Map and reduce functions in that approach operate on the local storage of each node, and for that reason the authors were forced to change the implementation of collective UPC functions to make them work with local memory space of each thread. In our implementation we used different approach based on shared hashmap data structure to perform key/value exchange. Hashmap instances reside in shared address space and each instance has an affinity to a single thread. Accordingly, every thread has an access to hashmap instance of any other thread. In different paper [4] authors presented a similar approach where they applied X-10 library implementation of hashmap data structure to store locally in each thread intermediate key/value pairs and then merge all the values to one thread. X-10 enabled Mapreduce merging procedure is poorly scalable since all data is moved to a single place and therefore such an approach possesses inherent limitations associated with processing and storage capabilities of a single node. In our approach we keep one instance of shared hashmap per thread such that each thread works on local portion of its own shared hashmap and other threads when needed could perform remote operations on that thread-local instance of shared hashmap. Hence, processing is not limited by resources of a single node and only requires efficient data exchange after finishing map phase. Additionally, this way we can control locality of operations on each instance of hashmap and as a result later on can optimize key distribution among threads for reduce stage. Shared hashmap allows to efficiently extract and write key/value pairs in average O(1) time complexity. Consequently, based on features of hashmap data structure we attempted to reduce overhead associated with searching and extracting keys.

3 Main Part

3.1 Mapreduce on UPC Framework

Presented in the paper Mapreduce on UPC framework aims to bring together programmability benefits associated with UPC model with advanced processing power of Mapreduce technique. Implications of such architectural solution is that it is become very convenient to be able to express complex Mapreduce logic in a more concise form of UPC - Mapreduce by using global memory abstraction.

In order to implement Mapreduce in UPC we first wrote code for shared hashmap data structure based on shared memory operations such as upc_memput, upc_memcpy, upc_alloc, upc_memget and other. Operations on shared hashmap are controlled by our API functions such as shared_hashmap_put, shared_hashmap_get, shared_hashmap_resize, shared_hashmap_remove.

To store key/value elements we created globally addressable array of shared hashmap instances with default blocking factor of one in shared address space. Such layout of shared array corresponds to one-to-one mapping of threads and hashmap array entries. Consequently, each hashmap is designed to store key/value elements that are local to the thread executing map functions (see Fig. 1).

Map and reduce functions are specified by the application developer and are passed as parameters to init_mapreduce function that launches and controls the entire processing cycle of Mapreduce execution.

Each thread is assigned a number of map tasks. Each map task operates on exactly one input file. Therefore, in order to avoid imbalance, before map phase runtime distributes files among threads in such a way that each thread has approximately the same proportion of input files.

After all map functions are finished their execution, the shuffle phase take place. The shuffle phase is divided into 2 main stages:

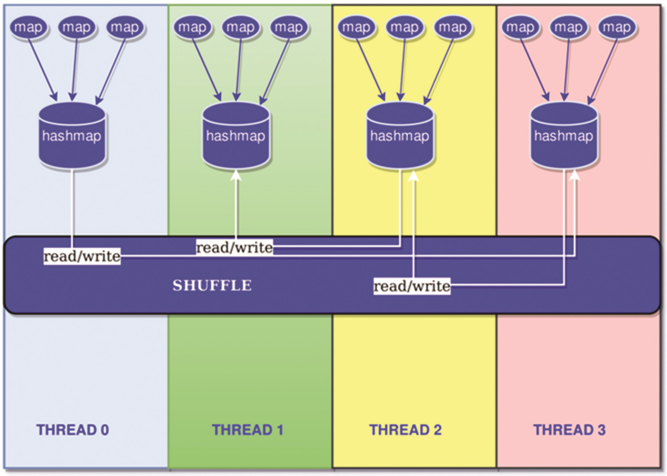

Fig. 1. UPC on Mapreduce map and shuffle design.

1. Data movement optimization and load balancing step.
2. Distribution of key/values among reducers.

In the process of load balancing we are using integer indexing of keys. We have to assign each key unique integer identifier. It is turn out that this operation is very expensive to perform since we need to traverse all hashmap entries in every thread by using only processing power of a single thread.

This thread is responsible for fetching remote hashmap entries and checking if that entry (key) already has been assigned identifier or not. If identifier already has been assigned to that entry (key) then we can skip it, otherwise it is required to update *id* field of that hashmap entry by remote write operation. Fetching and updating remote values by fine-grained operations incur a lot of communication and software overhead that should be avoided or substituted by coarse-grained bulk operations.

Hashmap element consists of the following fields: integer *id*, shared [] char * *key,* integer *in_use*, shared [] shared_vector * *data*. Since all field values are located in shared memory they can only be accessed by shared pointers. Shared pointers orders of magnitude slower than ordinary private pointers and therefore amount of accesses to shared memory area by shared pointers should be minimized.

Therefore, in order to minimize fine-grained access operations we developed more scalable and efficient in terms of running time method to assign each key unique integer identifier. We store keys in a shared array of string entries. A new method works by merging local to each thread keys stored in a shared array into a single shared array that has an affinity to thread number 0. The goal was to minimize number of copy operations. In a new method this number is equal to $O(\log n)$ compared to $O(n)$ operations in a previous implementation. There n represents number of threads.

```
function Merge (int $left$, int $right$,int $turn$,shared string * $keys$)
    Input: left index of array range $left$, right index of array range
           $right$, side to which data is copied $turn$,shared pointer to
           keys array $keys$
    $mid = left + (right - left)/2$ ;
    if $left < right$ then
        Merge ($left$,$mid$,0);
    end
    if $mid + 1 < right$ then
        Merge ($mid + 1$,$right$,1);
    end
    if $right - left \geq 1$ then
        if $turn = 0 \wedge MYTHREAD = left$ then
            Concat ($keys[left]$,$keys[right]$);
        end
        else if $turn = 1 \wedge MYTHREAD = right$ then
            Concat ($keys[right]$,$keys[left]$);
        end
    end
    Barrier;
```

Listing 1. Procedure for coarse-grained merge of key arrays

Merge procedure uses divide and conquer method that works according to Listing 1.

3.2 Data Movement and Load Balancing Optimization

For load balancing and data movement optimization we employ heuristic approach based on genetic algorithm [5]. Genetic algorithms are used in many problems in domain of combinatorial and multi-objective optimization. The problem with many instances of combinatorial optimization tasks is that they belong to NP class of problems. Therefore they cannot be solved by means of polynomial time algorithms and only hope to find a feasible solution for sufficiently large dimensions is to apply different heuristic approaches.

The following set of equations describes the problem:

$$min \sum_{i=0}^{threads-1} \sum_{j=1}^{keys} x_{ij} \times cost_{ij} \tag{1}$$

$$x_{ij} \in \{0, 1\} \tag{2}$$

$$min\left(\max_{i,j=0..threads-1} \left| load_i - load_j \right| \right) \tag{3}$$

$$load_i = \sum_{t=0}^{threads-1} \sum_{j=1}^{keys} x_{ij} \times size_{tj} \quad (4)$$

The optimization problem we have stated above is a modification of "Generalized assignment problem" which is known to be NP-hard. Genetic algorithms for solving GAP has been presented in different sources before, e.g. in [6, 7].

In order to find cost of assigning key j to thread i we construct cost matrix in which each entry $cost_{ij}$ is corresponding cost value of moving key j to thread i. Quantitatively, cost represents number of elements of some particular key that needs to be moved to some other thread. Formula (3) defines load balancing function. Load balancing function is calculated as minimum value over maximum difference of loads assigned to different pairs of threads. We need to perform distribution of key/values among threads with aim to optimize both functionals defined in formulas (1) and (3). Formula (2) defines the domain of x_{ij} variable to be consisting of two integer values of either 0 or 1. For thread i and key j the value of $x_{ij} = 0$ when thread i is not assigned to process key j and $x_{ij} = 1$ otherwise. Load value for each thread i is defined in formula (4). Genetic algorithm works according to following procedure:

```
function LoadBalance (int $n$, chromosone $p$, int $m$)
    Input  : Initial population $p$, Max number of generations $n$,
             Population size $m$
    Output: shared array $sol$
    $i = 1$;
    $np \leftarrow \emptyset$ ;
    while $i \leq n \vee$ stopping criteria is not met do
        ComputeFitness ($p$);
        for $j = 1$ to $m$ do
            $p1 \leftarrow$TournamentSelection($p$);
            $p2 \leftarrow$TournamentSelection($p$);
            $child \leftarrow$Crossover ($p1$,$p2$);
            Mutation ($child$);
            Enque ($child$,$np$);
        end
        $p \leftarrow np$;
        $np \leftarrow \emptyset$;
        $i = i + 1$;
    end
    $sol \leftarrow$SelectBestFitnessSolution ($p$);
    return $sol$ ;
```

Listing 2. Genetic algorithm for load balancing of keys among reducers

In order to be able to adapt genetic algorithm to solve our problem we first need to identify how to represent solution in the language of genetic algorithm. Solution (chromosone) is represented by vector, where *i*-th entry contains number of the thread that is assigned to process *i*-th key. Population is defined as set of all solutions and can be selected and correspondingly adjusted depending on specific needs and limitations of the task. Fitness value is an objective function that can be calculated for each particular solution. The task of genetic algorithm is to find specific solution with best fitness value. Fitness function in our problem is represented by combination of functionals described in (1) and (3).

Then, after genetic algorithm generates a solution, runtime can proceed to perform shuffle procedure.

3.3 Shuffle Phase

To perform shuffle procedure we need to appropriately distribute key/values among reducers such that each reducer can then schedule to perform reduce function calls on input elements with same key. In our program we have implemented shuffle procedure as follows:

- To store key/value elements on reduce side we created a new array of shared hashmap data structures with default layout in shared address space
- Each hashmap of the old array on each thread is traversed in parallel and according to the thread-keys mappings, obtained by solving optimization problem, elements are copied to threads that are assigned to process current element (key).
- After key/value distribution completes, each thread is ready to run reduce functions

Reduce stage is organized such that on each thread shared hashmap is traversed and each hashmap entry of <key, set of values> is assigned as input to a single reduce function. After completing their execution each reduce function writes final result to a single resulting file.

3.4 WordCount Implementation

For experimental evaluation of our Mapreduce framework we have chosen WordCount benchmark application. WordCount program computes number of occurrences of each word in a set of documents. This problem is a standard application for evaluating Mapreduce-based frameworks. The main idea behind implementing WordCount on Mapreduce is to divide processing such that each mapper emits for every word a pair of <word, 1> and each reducer then add all entries in the list of 1's that has been assigned to it and emits as final result pair of <word, overall_count>. In code listings 3 and 4 below our map and reduce function implementations for WordCount application are presented. The code for map and reduce functions must be written in C language with possible use of UPC-related functions for shared memory operations.

```
void * map (string filename)
{
   char * file_data;
   file_data = read_file_contents (filename);
   Vector tokens;
   vector_init(&tokens);
Tokenize (file_data,&tokens);
for (int i = 0;i<tokens.size;i++)
{
   collect (vector_get (&tokens,i),1);
}
free(file_data);
}
```

Listing 3. Implementation of map function for WordCount application

```
void reduce (string key,shared [] vector_sh
*values)
{
   int i;
   int cnt = 0;
   for (i = 0;i<values->size;i++)
   {
      int v = vector_get_shared_copy (values,i);
      cnt+=v;
   }
   reduce_collect (key,cnt);
}
```

Listing 4. Implementation of reduce function for WordCount application

4 Experimental Results

In this section we present results of evaluation of UPC on Mapreduce framework based on Google cloud platform architecture. The setup consisted of one instance of n1-highmem-8 (8 vCPUs, 52 GB memory). In our experiments we used the following software:

- Berkeley UPC runtime version 2.24.0
- Apache Hadoop version 2.7.3
- The Berkeley UPC-to-C translator, version 2.24.0

WordCount application has been tested for different input sizes ranging from 50 to 200 megabytes. Based on results of running WordCount on Apache Hadoop and UPC on Mapreduce (see Fig. 2) we can conclude that Mapreduce on UPC shows better performance on all inputs besides smallest 50 Mb input in which both frameworks show the same performance.

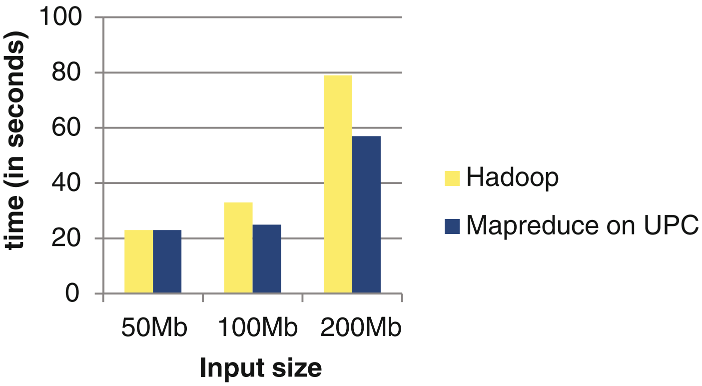

Fig. 2. Hadoop and Mapreduce on UPC running time for different input sizes

5 Conclusion

The paper presented UPC on Mapreduce framework that allows to users to implement data parallel applications by expressing them in the form of map and reduce functions. By analyzing results of evaluation of Mapreduce on UPC framework we observed better performance results compared to Hadoop, but algorithm have some scalability issues in case of small number of threads performing WordCount task.

References

1. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Sixth Symposium on Operating System Design and Implementation (OSDI2004), p. 10. USENIX Association, San Francisco (2004)
2. Carlson, W.W., Draper, J.M., Culler, D.E., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and language specification. Technical report, IDA Center for Computing Sciences (1999)
3. Teijeiro, C., Taboada, G.L., Tourino, J., Doallo, R.: Design and implementation of Mapreduce using the PGAS programming model with UPC. In: 17th International Conference on Parallel and Distributed Systems (ICPADS 2011), pp. 196–203. IEEE Computer Society, Washington (2011). doi:10.1109/ICPADS.2011.162
4. Dong, H., Zhou, S., Grove, D.: X10-enabled MapReduce. In: 4th Conference on Partitioned Global Address Space Programming Model (PGAS 2010), pp. 1–6. ACM, New York (2010). doi:10.1145/2020373.2020382
5. Man, K.F., Tang, K.S., Kwong, S.: Genetic algorithms: Concepts and applications. IEEE Trans. Industr. Electron. **43**(5), 519–534 (1996). doi:10.1109/41.538609
6. Chu, P.C., Beasley, J.E.: A genetic algorithm for the generalised assignment problem. Comput. Oper. Res. **24**(1), 17–23 (1997). doi:10.1016/S0305-0548(96)00032-9
7. Liu, Y.Y., Wang, S.: A scalable parallel genetic algorithm for the generalized Assignment Problem. Parallel Comput. **46**, 98–119 (2015). doi:10.1016/j.parco.2014.04.008