Implementing parallel algorithms of MapReduce

Grigory Fedyukovich[1], **Vladimir Safonov**[1]
[1]Department of Mathematics and Mechanics, Saint Petersburg State University, Saint Petersburg, Russia

Abstract— *In this paper we present the implementing of parallel algorithms of MapReduce problem in C++, C# and F# languages with technologies OpenMP, MPI, TPL, PPL, Threading. Firstly we describe the algorithm and introduce the basics of its parallelization. At the last section we show the execution results of parallelized programs.*

Keywords: MapReduce, parallelization, optimization, performance

1. Introduction

The papers "Future of Computer Architecture" by David Paterson [1] and "The Landscape of Parallel Computing Research: A View From Berkeley" of Berkeley University professors [2] were chosen as a prerequisite for the investigation. These papers propose the list of well-known and widely-applicable problems, also called "dwarfs", which use pattern-based computations and parallel processes communication technologies in their implementation. MapReduce [3] is one of such dwarfs.

The basic part of the research was done during the "Parallel Dwarfs" [4] project (`http://parallelldwarfs.codeplex.com/`). This project is based on two above mentioned papers. Its goals is to get academic and industrial software developers familiar with dwarfs and to implement the dwarfs concurrently in different programming languages, platforms, using different parallel programming technologies, and to compare the results in terms of performance and convenience of use.

2. The MapReduce method

Stable operation of a multiprocessor system requires to balance the workload of the processors. The first thing to be done is to find the key states in the algorithm - such ones that saving of intermediate data in those states is not only reasonable but also convenient. This data is already structured and there is not necessary to spend extra resources for the restructuring.

Such design improves the ability of the system to be fail-safe. In case of a program execution crash, it is necessary to start the execution again, i.e. to repeat some first actions for the source data. Now, if we group some actions logically, and collect their intermediate results, then there is no need any more to execute the corresponding actions next time - it is enough to start execution from the last saved point.

The second step after finding the key states is the analysis of technologic properties of the system: the performance of the processors, the speed of data transfer. We need to create a correspondence between the processors and sub-tasks of our algorithm, i.e. to start the execution of any part of algorithm, bounded by two neighbour states for each processor.

The MapReduce method consists of two parts: Map and Reduce. The source data of the method is a finite set S. It is the first key state. As a result of the method, we should obtain a set of pairs $D = \{ (S', c) \mid S' \subseteq S \}$. Here S' is a subset of S, satisfied by the condition P, c is the count of S'' (subsets of S), such as $S'' = S'$. The functionality of the Map sub-method is to apply P for each subset of S. The set of P-subsets (the list) is the saved intermediate data after performed termination of the Map. This storage indicates the second key state. Reduce, the sub-method that runs the next, finds and counts all equal sets. Its result represented by the dictionary is the last key state.

There are many interpretations of the problem where the MapReduce method is applicable. The area of data search has a classical problem of words counting in a given text. The input data set S for the problem in question is an ordered and indexed set of symbols, where the symbols with equal values but different indexes are not the same by definition. Words are connected subsets of S, containing only the letters as elements. Letters are elements of a pre-defined alphabet. The Map sub-function finds words in the text and returns the set of words represented as a list. Then Reduce sorts the list that leads to grouping of the words. It is enough for the algorithm exit to create the list of pairs (word, its entries count). This action can be performed just by one list traversal. The processes interaction scheme for the above is shown in Figure 1.

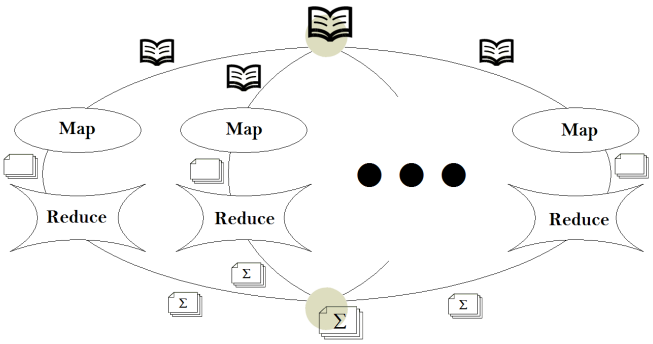

Fig. 1: MapReduce architecture

3. MapReduce implementation

3.1 OpenMP use with C++

Serial implementation of MapReduce can run on every N-processor system, where each i-processor contains K_i kernels. But the termination of a program will use only one kernel of one processor. Source data is not divided into sub-parts, Map produces only one list, and Reduce - one dictionary. In case of source data partitioning into n pieces and running for Map and then Reduce for each one, all of them will be compiled to the sequence of n calls.

OpenMP is the programming interface allowing us to use parallelism without changes of already written code. It offers a set of directives for parallelizing loops, locking and using private data structures. The specification of OpenMP is published in [5]. The code of parallelized implementation is shown below.

```
list<string> ptStringEntries;
                        // the result of Map
map<string, int> ptStringTotal;
                        // the result of Reduce
omp_lock_t lk;
                        // new lock declaration
omp_init_lock( &lk );
       // lock initialization (common for all)
#pragma omp parallel
  for private (ptStringTotal, ptStringEntries)
       // directive for loop parallelization
for (int i = 0; i < n; i++){

    Map(ptStringEntries, i);
                                  // run Map
    Reduce(ptStringTotal, ptStringEntries);
                                  // run Reduce
    omp_set_lock( &lk );
                                  // lock
    SumThreadResult(ptStringTotal);
                              // summarize method
    omp_unset_lock( &lk );
                                  // unlock
}
omp_destroy_lock( &lk );
                        // destroy lock (common)
```

For each i-iteration, a thread is created. Every thread operates with its own substring parallelly. First of all, it fills the list $ptStringEntries$ and then the dictionary $ptStringTotal$. It is illustrated in Figure 2.

The use of $\#pragma\ omp\ parallel\ for\ private(...)$ ensures the isolation of local objects for each loop iteration.

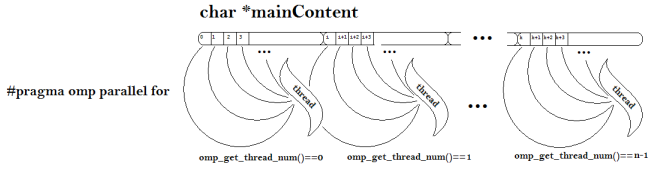

Fig. 2: Parallel data processing

The particular dictionary results have to be united together. This process should not be simultaneous, because it can cause conflicts. For prevention of it the synchronization mechanism is used. In OpenMP, there are locking operators - objects of the class omp_loct_t. There is only one lock for all threads. The initialization and destroy have to be terminated outside of the loop.

3.2 PPL use with C++

The Parallel Patterns Library (PPL) looks similar to OpenMP, from the viewpoint of its use. It was included into MS Visual Studio 2010. Basic classes, structures and operators of the library are defined in $Concurrency$ namespace specified in $ppl.h$ header file. The implementation principles of MapReduce are similar to the OpenMP ones, except for lambda-expressions use in parallel-executed loop body.

There are no locks used - this is yet another difference with OpenMP. We declare the subsidiary object of class $combinable$, which stores the local results of each thread (method combinable.$local()$) and then somehow unites (method combinable.$combine\_each()$). The refinement of unification method is also anonymous lambda-function. But note that it is possible to define the method outside and just call it from lambda:

```
combinable<map<string, int>> combinator;
parallel_for (0, n, [&](int i)
{
    list<string> localMapResult;
    map<string, int> ptStringTotal;
    Map(ptStringEntries, i);
    Reduce(ptStringTotal, ptStringEntries);
    combinator.local() = ptStringTotal;
            // store of intermediate results
});
combinator.combine_each
           ([&] (map<string, int>& local)
{                 // unite the local results
    SumThreadResult(local);
});
```

3.3 TPL use with C#

The Task Parallel Library (TPL) is analogue of PPL for .Net. Let's look for the C# example. The delegates of this languages are used as lambdas:

```
System.Threading.Parallel.For
                    (0, n, delegate(int i)
{
    var ptStringTotal = Reduce (Map(i));
    lock(commonResult)
    {
        SumThreadResult(ptStringTotal);
    }
}
```

The synchronization here is achieved through classic locks, which were introduced far earlier than TPL itself. Monitors: Monitor.$Enter(Object\ o)$ and

Monitor.$Exit(Object\ o)$ can be used here as the guards for operation with the $Object\ o$ as well.

3.4 System.Threading.Thread use with C#

The classical approach to the concurrent programming in .NET is the use of System.Threading.$Thread$ objects. This is a more complicated process, but the results show (see Testing section) that it is the most effective way of parallelization for the problem. The general approach of thread creation and run is presented below.

```
var threadStart =
         new ThreadStart(MapReduceMethod);
var mapReduceThread =
         new Thread(threadStart);
mapReduceThread.Start();
```

The body of the method executed by a thread is implemented separately in C# - in our case there is void $MapReduceMethod()$. This method is passed to the thread using the $ThreadStart$ delegate. However $MapReduceMethod$ does not accept any parameters, therefore there is no possibility to pass additional data through them. It means that for creating of n threads, we have to implement n methods, where everyone uses unique data. That is the reason of using of object models here. Let's create the class $MapReduceBaseMethod$ - the encapsulation of fields (input string, list of words, dictionary) and methods (Map, $Reduce$, $BeginMethod$). $BeginMethod$ here is just a sequence of calls of Map and $Reduce$.

```
var currentMapReduce =
        new MapReduceBaseMethod(mainContent);
currentMapReduce.BeginMethod();
Dictionary<String, int> stringTotal =
           currentMapReduce.GetStringTotal();
```

We defined void $BeginMethod()$ as a dynamic method, therefore it depends on particular $MapReduceBaseMethod$ object. It is enough to pass $BeginMethod$ to the $ThreadStart$ delegate while creating a new instance of this. Next, create an array of size n for the objects of $MapReduceBaseMethod$, where the $mainContent$ field will contain different strings. As a result we will have n different methods:

```
var currentData = new MapReduceBaseMethod[n];
for (int i = 0; i < n; i++){
  currentMapReduce[i] =
    new mapReduceBaseMethod
                  (mainContent.GetPart(i)));
     // new object with i-part of mainContent
  var threadStart =
    new ThreadStart
           (currentMapReduce[i].BeginMethod);
              // particular delegate creation
  var mapThread = new Thread(threadStart);
              // thread creation
  mapThread.Start();
              // thread run
}
```

3.5 System.Threading.ThreadPool use with C#

Thread pool is a methodology to support operations with threads. It does not require creation of each of the threads since it is possible to pass the parameters to the Map and Reduce methods. Therefore it is not necessary to use $MapReduceBaseMethod$ class. Methods of each thread can be defined using the delegates inside the pool.

```
var results[] = new Dictionary<String, int>[n];
          // array for local results storage
var signal = new AutoResetEvent(false);
             // new signal for synchronize
int counter = n;
                        // threads counter
for (int threadCurrent = 0;
             threadCurrent < n;
                  threadCurrent++){
  ThreadPool.QueueUserWorkItem
                     (delegate(Object o){
                   // addition of a thread
    int i = (int)o;
         // get the number of current thread
    results[i] = Reduce (Map(i));
         // run sequentially Map and Reduce
         // and strore the result
    if (Interlocked.Decrement
                     (ref counter) == 0){
         // while the counter more than 0,
      signal.Set();
         // pool is waiting for termination
    }
  }, threadCurrent);     // loop iterator
}
signal.WaitOne();
   // main is waiting for termination signal
SumThreadResult(results);
   // summarization without locking
```

Signals are used for synchronization between threads. It is simple events that operate by changing of the value of a subsidiary Boolean variable. A signal is initialized in the main thread by setting the false value to this variable. While the variable has not become true, the execution of main thread is sleeping. Therefore, the execution inside pool may take as long time as necessary, because changing the variable value occurs at the end of a thread. It is evident that if all threads execute independently, then the value should be changed by the last thread. Who it will be, the thread counter decides. After the termination of each thread body, the counter decrements. Since it is an atomic operation, we do not need to use heavyweight locking mechanisms. There is static class System.Threading.$Interlocked$ which contains atomic operations and guarantees the synchronization, as a consequence, makes the performance more efficient.

3.6 TPL use with F#

One more innovation in Microsoft Visual Studio is support of the novel F# functional programming language. Its the most important feature is the opportunity to use objects and import all necessary namespaces of .NET. It is possible to

use the same data structure as for C# implementations and the same parallelization techniques. Here we consider the TPL example. The principles of the usage are completely the same as for C#. The concurrency is entered the loop by the use of the operator Parallel.$For()$, the locks - by the operator $lock$, and the delegate of the thread body - by an instance of class $Action$. The iterator for the parallel loop (an integer variable) points to the corresponded part of pre-partitioned source data. We apply function Map ($|>$ map) to these data and Reduce ($|>$ reduce) to the results of the Map.

```
let ParallelTask i =
                  // a tpl task declaration
    let ptStringTotal =
        sourceData.[i] |> map |> reduce
    lock commonResult
        (fun () ->
            sumThreadResult(ptStringTotal))
Parallel.For
    (0, n, new Action<int>(ParallelTask))
                // tasks creation and run
```

3.7 Async use with F#

The Async technology is a classical approach to create and run threads in F#. It can be compared to the threading in C#. But in our particular problem there is no so difference with the TPL usage. Like in previous example firstly we define the thread body and then create instances of threads based on this definition and run all of them.

```
let task i = async {
                  // an async task declaration
    let ptStringTotal =
        currentDataArray.[i] |> map |> reduce
    lock commonResult
        (fun () ->
            sunThreadResult(ptStringTotal))
}
let tasks =
    [for i in 0 .. n - 1 -> task i]
                  // tasks creation
Async.RunSynchronously(Async.Parallel tasks)
                  // tasks run
```

3.8 Use of Message Passing Interface

The approach to use parallelism in systems without shared memory differs from the shown one. In general, we need to distinguish between concurrent computations and distributed systems. But in practice we can model multiprocessor systems on a single host and execute parallelized program on it. In Section 4, the performance results of execution of both concurrent and distributed programs performed on one host are presented. In this section we consider parallel implementation of the task in a cluster using Message Passing Interface (MPI).

MPI defines an API for the data exchange between processors, that does not depend on a platform. For Windows we use MPI.Net SDK 0.5.0 (for C# and F#) and MS.MPI (for C++). For example, for sending an array from one node of a cluster (A1) to another (A2), we need to call a command that depends on the node: schematically on A1: $Send(array, A2)$, and on A2: $Receive(A1, array)$.

In order to solve the MapReduce problem on the system of N processors, basically we need to define the functionality for each of the processor, and to perform the compiled programs separately on each processor. MPI allows to combine all the methods in one program and to run the same program on every processor in the system. The MPI environment assigns the identical numbers to the nodes and allows manipulating by the data between nodes using these identifiers.

For our algorithm, first of all, we choose the "main" node that gets the input string, partitions this into N parts, then sends the parts to the nodes and wait for receiving the result. The other nodes receive private data, run Map and Reduce and send the result back to the "main" node.

```
string dataPart = null;
  // declaration of an object for string part
if (MPI.Communicator.world.Rank == 0)
        // checking the "main" node
{
  int n = MPI.Communicator.world.Size;
        // count of processors
  string data = GetSource();
        // "main" node gets whole string
  string[] dataParts = GetParts(data);
        // then partitions it
  for (int i = 1; i < n; i++)
  {     // then sends to all nodes
     MPI.Intracommunicator.incomm.
       Send<string>(dataParts[i], i, root);
  }
  dataPart = dataParts[0];
        // points 0-part for own processing
}
else
{       // other nodes receive the data
  MPI.Intracommunicator.incomm.
          Receive<string>(0, 0, dataPart);
}
var mapResult = Map(dataPart);
        // run Map for the own part
var reduceResult = Reduce (mapResult);
        // run Reduce for the own part
//
// and next the place for code
// of sending results back (analogously)
```

Note that functions Map and Reduce will be processed in all nodes, even in the "main" one (denoted by identifier 0). It happens because there are no condition guards around Map and Reduce calls. We put the checking of the host identifier (MPI.Communicator.world.$Rank$) to the test, so if the execution of the program is performing at the state of condition, then the consequent will be executing only if the host satisfies to the test. The source string $dataPart$ is different for all the hosts because it

was received individually. As shown in example, "main" node do (MPI.Communicator.world.$Size$ - 1) sending operations, therefore it is necessary to do same count of receiving ones. Someone can ask, why there is only one receive operator, but we clarify that this code is performed on all processors in the system (i.e. these count is MPI.Communicator.world.$Size$) and the consequent of the condition (MPI.Communicator.world.Rank.$Size$ == 0) holds only once. Therefore for other (MPI.Communicator.world.$Size$ - 1) times the alternative holds and receive-operator performs.

4. Testing and results

The applications were built by MS Visual Studio 2010 Beta 2 and tested on the following hardware configuration: laptop with 3GB RAM, Intel Core2 Duo 2,26 HGz CPU, Vista SP2, Microsoft.NET Framework 4 Extended Beta 2; MS Visual C++ 2010 Beta 2 x86 Runtime, MPI.NET Runtime. For the cluster emulation the Microsoft HPC MPI Application Launch was used (mpiexec and smtp utilities).

Table 1: The average execution time (sec) for C++

source data	Serial	PPL	OpenMP	MPI
25 Mb	10.066	6.062	6.047	5.809
50 Mb	22.230	13.450	13.503	12.341
100 Mb	49.509	31.678	31.837	27.362

Table 2: The average execution time (sec) for C#

source data	Serial	TPL	Thread	ThreadPool	MPI.Net
25 Mb	12.243	8.317	7.416	7.308	7.812
50 Mb	26.813	17.372	16.112	15.910	16.183
100 Mb	57.125	37.484	35.869	35.995	33.784

Table 3: The average execution time (sec) for F#

source data	Serial	TPL	Async	MPI.Net
25 Mb	14.386	8.412	8.536	8.145
50 Mb	30.856	17.722	17.723	17.2641
100 Mb	66.640	37.924	38.193	37.130

5. Conclusions

A lot of algorithms we use in our programs can be parallelized by adding of one line of code. But programmers who do not know the advantages of parallelism, have to develop non-optimized code. We hope the illustrated work will be a good help while studying the concurrent programming. We showed the strengths and weaknesses of such technologies as TPL, OpenMP, MPI for C++, C# and F# with respect to MapReduce problem, but other implementations on different platforms and using different methods can also be developed and compared. The experience and positive efforts of such research give us a lot of opportunities and topics for oncoming investigation.

6. Acknowledgements

This research as well as the "Parallel Dwarfs" project were funded in part by a grant from the Microsoft Corporation. We a pleased to say many thanks for the people who worked with us under the project: Sean Mortazavi, Robert Palmer, Jeffrey Sax, Matej Ciesko, George Bosworth, Ilya Yuneev and Nikolay Viskov.

References

[1] David A. Patterson. Future of Computer Architecture 2006.

[2] Krste Asanovic Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb, Williams, Katherine A. Yelick. The Landscape of Parallel Computing Research: A View From Berkeley. University of California, 2006

[3] Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. Google, Inc. (c) 2004

[4] Sean Mortazavi, Jeff Baxter, "Building Supercomputer Applications using Windows HPC 2008", Microsoft, 2008

[5] OpenMP Application Program Interface. Version 2.5 May 2005, (c) 1997-2005 OpenMP Architecture Review Board.