BARRACUDA, AN OPEN SOURCE FRAMEWORK FOR PARALLELIZING DIVIDE AND CONQUER ALGORITHM

Abdourahmane Senghor

Department of Computer Science, Cheikh Anta Diop University, Dakar, Senegal

ABSTRACT

This paper presents a newly-created Barracuda open-source framework which aims to parallelize Java divide and conquer applications. This framework exploits implicit for-loop parallelism in dividing and merging operations. So, this makes it a mixture of parallel for-loop and task parallelism. It targets shared-memory multiprocessors and hybrid distributed shared-memory architectures. We highlight the effectiveness of the framework and focus on the performance gain and programming effort by using this framework. Barracuda aims at large public actors as well as various application domains. In terms of performance achievement, it is very close to Fork/Join framework while allowing end-users to only focus on refactoring code and experts to have the opportunity to improve it.

The barracuda main code implementation is available at https://github.com/boya2senghor/Barracuda

KEYWORDS

divide-and-conquer; task parallelism; parallel for-loop; Fork/Join framework; shared-memory multiprocessors

1. INTRODUCTION

In the context of multi/many cores era with an exponential growth, parallel thinking, learning, designing, programming, computing, debugging, and so on, should be the new paradigms. Supercomputer, grid computing, desktop computer, smartphones, embedded computer, and so on, are all equipped with multicore processors. Mathematical, thermodynamics, climate science, aero-spatial, graphical processing applications, and so on, are all computing-intensive. Operating systems (Windows, Linux, Unix,..), runtimes and system executors (gcc, jre,…), languages (c/c++, java, matlab, cilk, fortran), compilers and built-in API (openMP, MPI) all provide with facilities to leverage the underlying multicores architecture. However, most scholar academic, developers, companies still continue focusing their effort in sequential learning, programming, computing paradigms.

Now, the actual and great challenge is to develop efficient strategies in order to convince those communities to adopt parallel programming paradigm. This one basically includes data (or loop) parallelism and task (method) parallelism algorithms. The main papers, tools, frameworks mostly focus on the first one. Nevertheless, in this paper we approach the task parallelism specially the divide and conquer algorithm.
Many applications (mergesort, quicksort, dichotomy searching, Fibonacci, integrate, strassen matrix multiplication,…) in various scientific fields (numerical sorting and searching, mathematical calculation, image processing, etc.) rely on divide and conquer algorithm.

David C. Wyld et al. (Eds): DBML, CITE, IOTBC, EDUPAN, NLPAI -2023
pp. 63-75, 2023. IJCI – 2023 DOI:10.5121/ijci.2023.120206

We use Java as target language, compiler and runtime. Java is the most widespread platform ranking from supercomputers to smartphones. Java supplies as well primitives as advanced libraries and API for developing parallel programs. For non-expert parallel programmer, one of the best way to deal with divide and conquer is to use high level API such as Fork/Join framework [7][9]. Using primitives API such as threads, runnable, synchronization is a domain reserved for experts.

In this paper, we propose a newly-created open source framework, we name Barracuda. This framework is purely built around Java primitives API and is aimed at both end-users and experts. End-users focus their effort simply in the copy-past-replace-comment out action according an operating mode. He does not need to deeply know the concept of parallel programming. The expert, however, may reuse this framework and improve it, in terms of performance outcome and build high level API.

The core implementation of the framework consists of more than nine (9) functions and two (2) internal classes. The functions are classified in three functionalities which are dividing, processing and merging. The first internal class encapsulates the parameters and result of the current recursive method in the same object. The second one implements a runnable object, we call task.
Task parallelism relies on light thread management. Furthermore, work stealing algorithm is used in order to avoid a thread being idle while there are some tasks remaining in certain queues. That strategy improves the overall performance. Parallelizing compilers like openMP [1][3] supplies task parallelism , API and environment variables.

Intel TBB[6][10] supplies API to deal with divide-and-conquer algorithm. In habaneroJava[4], the asynch/finish concept is used to parallelize divide-and-conquer. HabaneroJava is an extension of the Java language and is introduced as a course at Rice University.

In [2], authors implement a framework relying on TBB, OpenMP and TaskFlow which deal with divide-and-conquer, aiming to reduce programming effort and improving performance.

One of the difficulties to be addressed with parallel recursive application is how to leverage the hybrid distributed shared-memory multiprocessors architectures. Most of the frameworks [27][28] are fitted to only target shared-memory multiprocessors architecture. Nevertheless, some frameworks like in [29], are designed and implemented to exploit both shared memory and cluster architectures. Our framework, in the fact that it explicitly divides task in subtasks which are stored in an array, potentially offers the possibility to schedule and distribute subtasks over nodes of cluster or hybrid distributed shared-memory architecture.

The sequential consistency, performance gain and programming effort are the core of the parallel programming. In our framework, when the instructions are strictly followed, it guarantees sequential consistency that is a necessary but not sufficient condition. However, it’s not worth parallelizing an application if we may not obtain performance gain compared to sequential one. Our framework allows performance gain and tends to reduce programming effort.

2. FRAMEWORK DESCRIPTION

2.1. Framework Architecture and Implementation

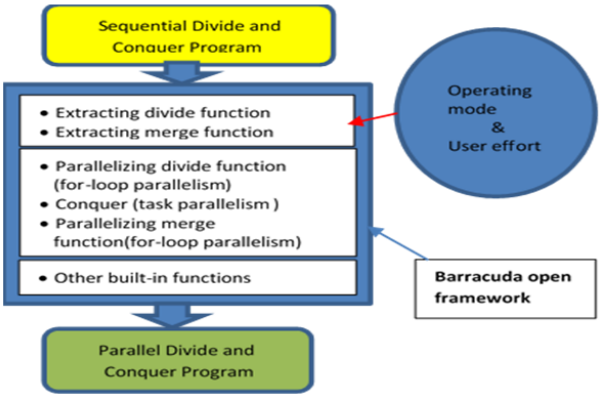

Figure 0: Barracuda Architecture

The main architecture of Barracuda highlights the process of extracting dividing and merging function from the input sequential recursive method. End-user and programmer focus their effort in that extracting process. Barracuda provides with for-loop parallelization for those functions. Other built-in functions and internal classes make up the framework. No intervention from the end-user is required for those functions.

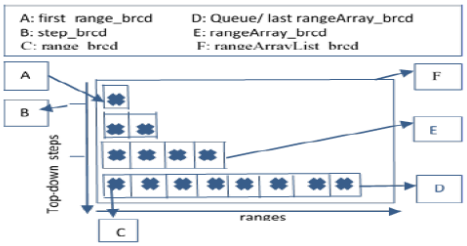

Figure 1.a: Top-Down Matrix of ranges perspective

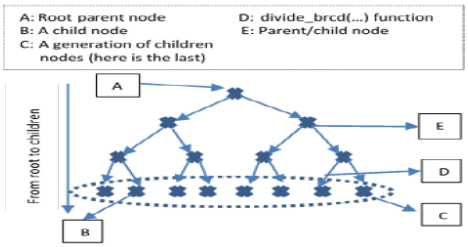

Figure 1.b: Tree of ranges 'perspective

The second and third figures (Fig1.a, Fig1.b) describe the process of dividing a 2-way recursive method.

- **range_brcd** is made up of recursive method 's formal parameters and result (if so) variables. In general purpose algorithm[2], problem (parameter) an result (return, if so) are managed separately. We decide to encapsulate problem and result in the same object we name range_brcd. From a matrix perspective, it represents an element or cell. From a tree perspective, it represents a node (parent or child).

- **step_brcd**: from a matrix perspective, it represents a row index and from tree perspective, it represents the level of a same generation of nodes (parents or children).

- **rangeArray_brcd**: from a matrix perspective, it represents a row of ranges. From a tree perspective, it represents a generation of nodes.

- **rangeArrayList_brcd**: from a matrix perspective, it represents itself the matrix (two-dimensional array). From a tree perspective, it represents the tree itself or a set of node generations.

- The framework skeleton is built around more than nine (9) functions and two (2) internal classes. However, parallel dividing and parallel merging supply additional functions and internal classes.

- **divide_brcd(…):** In the framework, this function has empty content and must be customized by user. The content of the recursive method will be copied into this function. The called recursive method will be replaced by operation of adding generated range_brcd into rangeArray_brcd of the next step_brcd (equal current step_brcd+1 position).

- **finalDivide_brcd(…):** This function traverses the matrix and for each step, traverses the columns, and for each cell divides it into **k_way_brcd range_brcd** (into k ranges). In addition, these ranges are added to the **rangeArray_brcd** array corresponding to "**step_brcd+1**". At the end, this function generates a triangular matrix of ranges. To divide each range, this function calls **divide_brcd**. The way to divide each range depends on the recursive method. The **numOfTasks** specified by the end-user determines the **cut-off** parameter and impacts the number of the steps in dividing process:

- **int numOfSteps_brcd = (int) (log(numOfTasks_brcd) / log(k_way_brcd))**; This function is built by the framework. No effort (refactoring) is needed from the programmer.

- **parallel_finalDivide_brcd (…)** is the parallel implementation of **finalDivide_brcd(…)**. Parallelization is done in phases. This function is built by the framework. No effort (refactoring) is needed from the programmer.

- **setTasks_brcd(…)** The **rangeArray_brcd** corresponding to the last **step_brcd** position is considered (represented by queue) . This array will be considered as a global resource or queue to be shared by next threads **workThread_brcd**. A **cursor_brcd** implementing atomicInteger counter (which is a non-blocking synchronisation) allows safely moving from an index of that array to another. This function creates as many tasks as there are **workerThread_brcd**. This defines how our queue is managed. Compared with work stealing queue [11] [12] [13] [14] management where each processor has its own queue, we have a unique queue shared among processors. One of the goal of work stealing is to avoid that a processor after ending up running its own queue, stay idle while some processors have tasks remaining in their queue. In our sharing queue implementation, as long as tasks remain in the queue, no processor will be idle. From this perspective, those two strategies produce the same result. We instantiate as tasks as the number of threads specified by the end user or by default the underlying runtime. We use primitive threads for processor execution and runnable interface for task implementation. This function is built and fully implemented by the framework. No effort is required from the programmer.

- **forkJoin_brcd(…):** for each task_brcd previously created, this function maps a workerThread_brcd. The main thread launching this function starts the different workerThread_brcd and waits for them until they end their execution. Here we use primitive fork-join synchronisation. This function is a built by the framework. No effort is needed from the programmer.

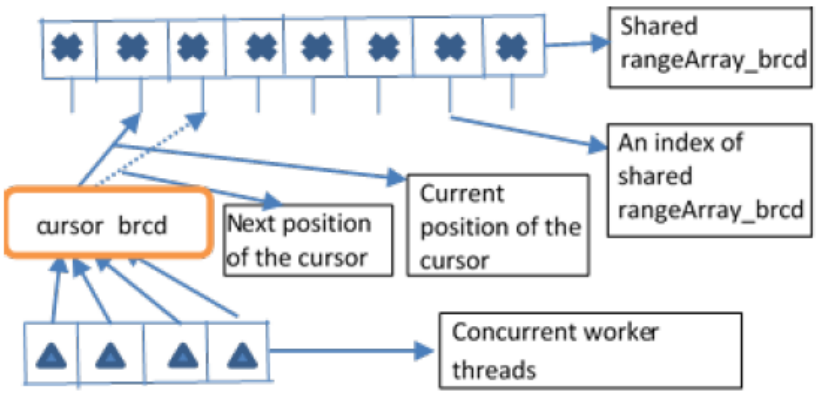

Figure 2: Shared queue management

• **merge_brcd(…):** Like **divide_brcd**, this function needs to be defined by the user. He simply replaces the called recursive method by the operation of recovering result in, for each range_brcd. This operation is made easier by the fact that we encapsulate parameters and result in the same range_brcd. We describe the effort of the programmer to customizing this function later.

• **finalMerge_brcd(…)**: this function is a bottom-up operation where, by a group of k_way_brcd, it merges their result and put it in their up parent corresponding range_brcd. This function traverses the matrix from bottom-up and iterates over the rows of the matrix. For each row, the k_way_brcd adjacent range_brcd are taken, in parameters by the merge operation and the result is post (composed) in their parent range_brcd result. At the end, the final result (the rangeArray_brcd or the original range_brcd) is obtained. This function is built by the framework. No effort (refactoring) is needed from the programmer.

• **parallel_finalMerge_brcd(…)** is the parallel implementation of finalMerge_brcd((…). Parallelization is done in phases. No effort (refactoring) is required from the end-user/programmer.

• **compute_brcd(…):** this function is the parallel function of the recursive method. It calls all method previously defined. range_brcd is the original actual parameter of the recursive function. numThreads_brcd is the number of worker threads to be executed. numOfTasks_brcd is a number of tasks the user wants the problem be divided up before starting task parallel execution. k_way_brcd is the k_way recursive method invocation.

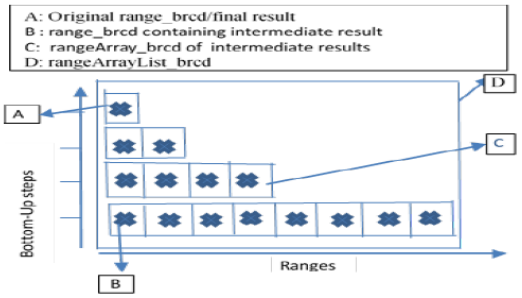

Figure 3.a: Bottom-up Matrix of ranges

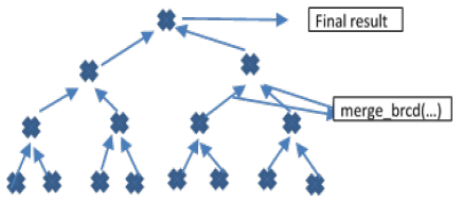

Figure 3.b: Bottom-up Tree of ranges

Fig3.a and Fig3.b describe the process of merging a 2-way recursive method.

2.2. Framework Functionalities

The framework presents the following functionalities:

- Transforming a sequential code to parallel one that can leverage the shared-memory multiprocessor architecture.
- Transforming a pure task parallelism code into a mixture of for-loop parallelism and recursive task-parallelism code.
- Running the parallel code by having the ability to influence the performance via the number of partitions obtained from dividing operation.
- In the context of hybrid distributed architectures which nodes are shared-memory multiprocessor, resulting tasks from dividing operation are distributed to different nodes.
- Giving the user the opportunity to customize the parallel code and add additional code.

2.3. Operating Mode and programmer Effort

The programming effort is reduced to **copy-past-replace** operation. Programmer does not need to be parallel programmer expert.

a) **Copy** the content within the block of **DivideAndConquer_BRCD** class and insert it just after the recursive method block.
b) **Copy and paste** formal parameters and (if so) return variable declarations of the recursive method within the Range_brcd class. Declare a “Range_brcd() “constructor with recursive method formal parameters as formal parameters;
c) Within “run()” method of “Task_brcd” class , **call** recursive method with actual parameters preceded by "rangeArray_brcd[i]." ; If the recursive method returns, prepend “rangeArray_brcd[i].result=” to the statement;
d) **Copy and paste** the content of k-way recursive method content within divide_brcd() function; prepend "range_brcd." to the original formal parameters by replacing “return [variable]” by return. Replace any called recursive method by " rangeArray_brcd[step_brcd][++indexOfRange_brcd]=new Range_brcd (...))". Comment out the following code immediately after the last called recursive method.
e) **Copy and paste** content of k-way recursive method within merge_brcd() function. Prepend "range_brcd." to the original formal parameters and replace “return” by "range_brcd.result=". Replace any called recursive method by " rangeArrayList_brcd[step_brcd][++indexOfRange_brcd].result”;
f) **Comment out** the called main recursive method. Declare a “new range_brcd(…)” with the actual parameter of the called recursive method; **Call** the “compute_brcd(…)” method with range_brcd , number of threads, number of tasks and k-way actual parameters. Set “range_brcd.result” into the “result” variable.

3. EFFECTIVENESS AND IMPACT

3.1. Illustrative Example

In this example, we illustrate framework’s application upon Fibonacci. Only functions and classes to be modified are shown. First, download the framework from *https://github.com/boya2senghor/Barracuda*. Follow instructions from section 2.3. The classes and functions to be modified are **static class Range_brcd{…}**, **class Task_brcd implements Runnable{…}**, **public void divide_brcd(..){…}, public void merge_brcd(…){…}.**

```
/*package and import*/
public class Fibonacci {
   public long fibonacci(int n) {
      if (n == 0)  return 0;
      if (n == 1)   return 1;
      long x, y;
      x = fibonacci(n - 1);
      y = fibonacci(n - 2);
      return x + y;
   }
   /** Operating mode : a)    **/    (Follow instruction a) from section 2.3)
     static class Range_brcd {
/** Operating mode : b)    **/       (Follow instruction  b) from section 2.3)
      int n;
      long result;
      Range_brcd(int n){
         this.n=n;
      }
   }
   class Task_brcd implements Runnable {
    /*code*/
      public void run() {
         try {
            while (true) {
               int i = cursor_brcd.getAndIncrement();    //cursor management by non-blocking synchronization
               /** Operating mode : c)    **/       (Follow instruction from c) in section 2.3)
              rangeArray_brcd[i].result=fibonacci(rangeArray_brcd[i].n);
            }
         } catch (Exception e) {
         }
      }
   }
public void divide_brcd(Range_brcd range_brcd, Range_brcd[][] rangeArray_brcd, int step_brcd, int indexOfRange_brcd)
{
/** Operating mode : d)    **/      (Follow instruction d) from section 2.3)
       if (range_brcd.n == 0)  return;
      if (range_brcd.n == 1)    return;
      //long x, y;
      //x = fibonacci(n - 1);  //this statement is commented out and replaced by the following statement
      rangeArray_brcd[step_brcd][++indexOfRange_brcd]=new Range_brcd(range_brcd.n - 1);
      //y = fibonacci(n - 2);  //this statement is commented out and replaced by the following statement
      rangeArray_brcd[step_brcd][++indexOfRange_brcd]=new Range_brcd(range_brcd.n - 2);
     //return  x + y;          //this statement is commented out
   }
   public void merge_brcd(Range_brcd range_brcd, Range_brcd[][] rangeArrayList_brcd, int step_brcd, int
indexOfRange_brcd) {
/** Operating mode : e)    **/ (Follow instruction  e) from section 2.3)
      if (range_brcd.n == 0)  {range_brcd.result=0; return ;}
      if (range_brcd.n == 1)  {range_brcd.result=1; return ; }
      long x, y;
      //x = fibonacci(range_brcd.n - 1);
      x=rangeArrayList_brcd[step_brcd][(++indexOfRange_brcd)].result;
      //y = fibonacci(range_brcd.n - 2);
      y=rangeArrayList_brcd[step_brcd][(++indexOfRange_brcd)].result;
      //return x+y
      range_brcd.result=x + y;
   }
/*Other required Barracuda method which do not need any modifications*/
   public static void main(String args[]) {
      final int n = 45;
      Fibonacci fib = new Fibonacci();
/** Operating mode : f)    **/ (Follow instruction f) from section 2.3)
      //long z = fib.fibonacci(n);
      Range_brcd range_brcd=new Range_brcd(n);
      fib.compute_brcd(range_brcd, 4, 32, 2);
      long z=range_brcd.result;
   }

}
```

3.2. Performance Evaluation

We evaluate and compare the performance of our framework with fork-join framework by running quicksort, mergesort, fibonacci, strassen matrix multiplication, integrate programs over an IBM System X 3650 with the following characteristics:

- 8 cores processors , 8 GB memory;
- Fedora 25 operating system;
- Java 1.8.0 runtime

Each of these programs is both parallelized using Barracuda and ForkJoin. An experiment execution consists of running one parallel program in target environment. We repeat 10 times each experiment and consider the average execution time. It's also important to note that for each program the best cut-off [15] [16] [17] [18] [19] [20] [21]is retained. At the end of the experimentation, we present the outcomes in the following tables.

Table 1. Barracuda vs. ForkJoin performance over Quicksort

Quicksort				
	workload	**Cut-Off**	**Execution time(ms)**	**Performance**
Barracuda	N=320x10^6	log2 (numOfTasks)=10	4862,0	**lost: -7%**
ForkJoin		(right-left)<300x10^3	4512,0	

Table 2. Barracuda vs. ForkJoin performance over Mergesort

Mergesort				
	Workload	**Cut-Off**	**Execution time(ms)**	**Performance**
Barracuda	N=160M	log2(#tasks)=4	6084,0	**lost:-2%**
ForkJoin		(right-left)<4999999	5979,0	

Table 3. Barracuda vs. ForkJoin performance over Fibonacci

Fibonacci				
	Workload	**Cut-Off**	**Execution time(ms)**	**Performance**
Barracuda	N=50	log2(#tasks)=18	20877	gain=+2%
ForkJoin		(n<30)	21305	

Table 4. Barracuda vs. ForkJoin performance over Strassen

Strassen Matrix Product				
	Workload	**Cut-Off**	**Execution time(ms)**	**performance**
Barracuda	N=4096	log7(#tasks)=1	12556	gain=+4%
ForkJoin		(cut_depth=4)	13089	

Table 5. Barracuda vs. ForkJoin performance over Strassen

Integrate				
	workload	**Cut-Off**	**Execution time(ms)**	**Performance**
Barracuda	start=-59 end=60	log2(#tasks)=10	4406	gain=+14%
ForkJoin		(cut_depth=13)	5103	

For quicksort and mergesort, ForkJoin outperforms Barracuda. However, for Fibonacci, Integrate and Strassen Matrix product, Barracuda outperforms ForkJoin.

To analyse the performance differences, many parameters must be taken in account:

- Phaser(barrier synchronization)[25] [26]/asynchronous execution;
- Task/data locality [22] [23] [24] ;
- Queue sharing/ work stealing.

Table 6. Barracuda vs. ForkJoin

	Barracuda	**ForkJoin**
Divide	Phase	Asynchronous
	Parallel [enable/disable]	Parallel
Queuing	Queue sharing	Private queue
Queue synchronization	Index Atomic Increment(non-blocking synchronization)	Work stealing
Conquer	Parallel execution	Parallel execution
Merge	Phase	Phase
	Parallel [enable/disable]	Parallel

Dividing operation: One of the main differences between Barracuda and Fork/Join framework resides in the dividing phase. Since Fork/Join uses task parallelism, as soon as a task is forked, it's is immediately executed. In contrast, our framework imposes a rendezvous synchronisation (barrier). Another difference is that, our framework supplies the ability to use sequential or parallel execution in the dividing operation. To do that, go to "compute_brcd(…)" method and comment/uncomment out parallel_finalDivide_brcd(…) / finalDivide_brcd(…). The reason is that some divide and conquer programs spend slight execution time during dividing process (such as Mergesort, Strassen, Fibonacci, Integrate). In contrast, quicksort spends more time during process dividing. For optimisation, end-user can decide to enable parallel dividing in quicksort. Meanwhile, it can decide to disable parallel dividing in the context of mergesort execution.

Conquer process: Here, queue management is critical. Barracuda uses a unique shared queue of ranges (data structure made-up of actual parameters of recursive method), but threads access the tail without blocking synchronisation because atomic index increment is enforced. This technique avoids a thread remaining idle while resource (range) is available at the tail. It achieves the same result than work stealing even thought their techniques are different.

Table 7: Memory-bound vs. Cpu-bound applications

	Memory-bound	Cpu-bound
Quicksort	+	
Mergesort	+	
Strassen	+	
Fibonacci		+
Integrate		+

Merging operation: As well Barracuda as Fork/Join framework requires processing in phase to achieve merging operation. As in dividing operation, Barracuda supplies ability to enable/disable parallel merging operation. For instance, quicksort does not require merging phase, Fibonacci spends slight execution time during process merging operation.

Table 8. Divide/Merge computing application

	Divide	Conquer	Merge
Quicksort	+	+	
Mergesort		+	+
Strassen		+	+
Fibonacci		+	
Integrate		+	

From the previous analysis tables, Barracuda seems to be more suitable for cpu-bound application than memory-bound application compared with Fork/Join framework.

3.3. Programming Effort Evaluation

It's very difficult to quantify the programming effort. Manually parallelizing an application requires understanding:

- Coding sequential application;
- Object -Oriented approach in the context of Java;
- Multithreading and synchronization;
- Core processors architecture, memory and cache hierarchy;
- Compiler optimisation;
- Etc.

Frameworks (Fork/Join, intel TBB, Open MPI[8]) as well as source-to-source parallelizing compilers (OpenMP) , automatic compilers (Intel C++[5]) aim to hide complexity of parallel programing by hand and to lessen this error-prone exercise. An interesting question is, for instance, how openMP is [more or less] easy to use than Intel TBB. Being aware that parallel programming is a trade-off of [less or more] effort and [less or more] gain performance, the expected response is not a logic one (yes or no). It deserves more analysis by taking in account the different actors (student, experts, researchers, engineers, etc.). For instance, a programmer on production enterprise should prefer using OpenMP than manual parallel programming. An expert should prefer manual programming than using openMP. By comparison, desktop end-user prefers further graphic interface than command line interface while linux expert prefers command line interface. The more, the level is low, the more we can leverage performance and the more the exercise is error-prone.

In our context, Barracuda is written using Java primitive threads, synchronisations, but end-users may only focus on refactoring code (copy, paste, replace, comment out) effort. In the illustrative example, in previous section, we refactor 5-10% of the code in terms of code lines.

A mathematical end-user when computing parallel Fibonacci application just follows the instructions as indicated in operating mode section. A parallel programming expert can, in the context of quicksort, for instance, sorts the queue before conquer process, in order to improve the global performance execution time. Moreover, by sorting the shared tail, by first firing the biggest ranges (difference between end and start indexes), the final execution time is considerably improved.

4. CONCLUSIONS

Our newly-created open source framework Barracuda is a contribution to parallelizing divide and conquer application in the context of Java. It is written in a pure Java code. It exploits implicit data-parallelism in dividing and merging operations, making it a mixture of parallel for-loop and task parallel framework. It provides sequential/parallel dividing, parallel conquer and sequential/parallel merging operations. Barracuda aims at large public from academic student learning parallel programming, programmer developing parallel application as well as large application domains such as mathematics, physics, astronomic, which all need more and more computing resources. This open framework lets non-expert end-user to only focus on refactoring code line and does not matter about parallel programming knowledge. Meanwhile, expert or researcher may focus on the core of this framework in order to improve it.

Future work: to make this framework further easier to use, we outlook to introduce:

- pre-processor annotation. This allows end-user, for instance, inserting a "@divide" instead of doing "operating mode d)";
- compiler directive;
- lamda expression.

To improve performance achievement, we outlook to supply sorting shared queue (if so) in order to first run the biggest ranges. This allows reducing conquer execution time. To make the framework exploiting more and more resources computing and targeting distributed memory architecture such as local area network, we outlook to insert java socket and rmi;

REFERENCES

[1] Lee, S.hyun. & Kim Mi Na, (2008) "This is my paper", ABC *Transactions on ECE*, Vol. 10, No. 5, pp120-122.

[2] Gizem, Aksahya & Ayese, Ozcan (2009) *Coomunications & Networks*, Network Books, ABC Publishers.

[3] Adams, Joel, & Elizabeth Shoop, (2014) "Teaching shared memory parallel concepts with OpenMP." *Journal of Computing Sciences in Colleges*, 30.1 : 70-71.

[4] Danelutto, Marco, et al., (2016) "A divide-and-conquer parallel pattern implementation for multicores."In *Proceedings of the 3rd International Workshop on Software Engineering for Parallel Systems*. ACM.

[5] Duran, Alejandro, et al., (2009) "Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp." *Parallel Processing. ICPP'09*. International Conference on. IEEE.

[6] Imam, Shams, and Vivek Sarkar, (2014) "Habanero-Java library: a Java 8 framework for multicore programming." In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*. ACM.

[7] Automatic Parallelization with Intel Compilers. *Available at: https://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers*
[8] Kim, Wooyoung, and Michael Voss, (2011) "Multicore desktop programming with intel threading building blocks." IEEE software 28.1: 23-31.
[9] D. Lea, (2000) "A Java fork/join framework." In *Proceedings of the ACM 2000 Java Grande Conference*. :36-43.
[10] Open MPI. *Available at: https://www.open-mpi.org/*
[11] SENGHOR Abdourahmane, and Karim KONATE,(2013) "A Java Fork-Join Framework-based Parallelizing Compiler for Dealing with Divide-and-conquer Algorithm." *Journal of Information Technology Review* 4.1 : 1-12.
[12] Intel Threading Building Blocks, (2017). *Available at: https://www.threadingbuildingblocks.org/*
[13] Berenbrink, P., Friedetzky, T., & Goldberg, L. A. (2003). The natural work-stealing algorithm is stable. *SIAM Journal on Computing*, 32(5), 1260-1279
[14] Dinan, J., Larkins, D. B., Sadayappan, P., Krishnamoorthy, S., & Nieplocha, J. (2009, November). Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (pp. 1-11).
[15] Chase, D., & Lev, Y. (2005, July). Dynamic circular work-stealing deque. *In Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures* (pp. 21-28).
[16] Acar, U. A., Charguéraud, A., & Rainey, M. (2013, February). Scheduling parallel programs by work stealing with private deques. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming* (pp. 219-228).
[17] Fonseca, A., & Cabral, B. (2017). Evaluation of runtime cut-off approaches for parallel programs. In *High Performance Computing for Computational Science–VECPAR 2016: 12th International Conference, Porto, Portugal, June 28-30, 2016, Revised Selected Papers 12* (pp. 121-134). Springer International Publishing.
[18] Duran, A., Corbalán, J., & Ayguadé, E. (2008, November). An adaptive cut-off for task parallelism. *In SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (pp. 1-11). IEEE.
[19] Fonseca, A., & Cabral, B. (2018). Overcoming the no free lunch theorem in cut-off algorithms for fork-join programs. *Parallel Computing*, 76, 42-56.
[20] Fonseca, A., Lourenço, N., & Cabral, B. (2017). Evolving cut-off mechanisms and other work-stealing parameters for parallel programs. In *Applications of Evolutionary Computation: 20th European Conference, EvoApplications 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings, Part I 20* (pp. 757-772). Springer International Publishing.
[21] Iwasaki, S., & Taura, K. (2016, September). A static cut-off for task parallel programs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation* (pp. 139-150).
[22] Duran, A., Corbalán, J., & Ayguadé, E. (2008). Evaluation of OpenMP task scheduling strategies. In *OpenMP in a New Era of Parallelism: 4th International Workshop, IWOMP 2008 West Lafayette, IN, USA, May 12-14, 2008 Proceedings 4* (pp. 100-110). Springer Berlin Heidelberg.
[23] Iwasaki, S., & Taura, K. (2016, September). Autotuning of a cut-off for task parallel programs. In *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)* (pp. 353-360). IEEE.
[24] Wang, K., Zhou, X., Li, T., Zhao, D., Lang, M., & Raicu, I. (2014, October). Optimizing load balancing and data-locality with data-aware scheduling. In *2014 IEEE International Conference on Big Data (Big Data)* (pp. 119-128). IEEE.
[25] Zhang, X., Feng, Y., Feng, S., Fan, J., & Ming, Z. (2011, December). An effective data locality aware task scheduling method for MapReduce framework in heterogeneous environments. In *2011 International Conference on Cloud and Service Computing* (pp. 235-242). IEEE.
[26] Ceballos, G., Hagersten, E., & Black-Schaffer, D. (2016). Formalizing data locality in task parallel applications. In *Algorithms and Architectures for Parallel Processing: ICA3PP 2016 Collocated Workshops: SCDT, TAPEMS, BigTrust, UCER, DLMCS, Granada, Spain, December 14-16, 2016, Proceedings (pp. 43-61)*. Springer International Publishing.
[27] Shirako, J., Peixotto, D. M., Sarkar, V., & Scherer, W. N. (2008, June). Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22nd annual international conference on Supercomputing* (pp. 277-288).

[28] Shirako, J., Peixotto, D. M., Sarkar, V., & Scherer, W. N. (2009, May). Phaser accumulators: A new reduction construct for dynamic parallelism. In *2009 IEEE International Symposium on Parallel & Distributed Processing* (pp. 1-12). IEEE.
[29] Eliahu, D., Spillinger, O., Fox, A., & Demmel, J. (2015). *Frpa: A framework for recursive parallel algorithms. University of* California at Berkeley Berkeley United States.
[30] Danelutto, M., De Matteis, T., Mencagli, G., & Torquati, M. (2016, October). A divide-and-conquer parallel pattern implementation for multicores. In *Proceedings of the 3rd International Workshop on Software Engineering for Parallel Systems* (pp. 10-19).
[31] González, C. H., & Fraguela, B. B. (2017). A general and efficient divide-and-conquer algorithm framework for multi-core clusters. *Cluster Computing*, 20, 2605-2626.

AUTHORS

Abdourahmane SENGHOR holds a computer science Phd degree since 2014 especially in parallel and distributed programming and computing. He conciliates research activity in academy world and Informatics department manager in enterprise world.