IFERP
connecting engineers... developing research

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Vol 11, Issue 1, January 2024

Framework for Parallel Programming Model in Bulk Data Operations

[1]Olubukola D. Adekola, [2]Oludotun Oluyade, [3]Oyebola Akande, [4]Wumi Ajayi, [5]Adesoji Adegbola

[1] [3] [4] [5] School of Computing and Engineering Sciences, Babcock University, Ogun State, Nigeria,
[2] Researcher, Pearlsoft Nigeria Limited, Lagos State, Nigeria
Corresponding Author Email: [1]adekolao@babcock.edu.ng, [2]oluyadeo@babcock.edu.ng, [3]akandeo@babcock.edu.ng, [4]ajayiw@babcock.edu.ng, [5]adegbolaa@babcock.edu.ng

***Abstract*— Although hardware manufacturers are relentlessly building multi-cores system, software parallelization will not advance simply because the hardware has been built that way; it is a conscious effort of developing frameworks and models that can optimally exploit inherent hardware parallelism and make software implementation easy. Thus, parallel programming is an essential concern of software engineers in order to meet up with the call -in technological development. In the generation where there is incredibly massive bulk data to deal with, software developers need to seek algorithms or techniques that will directly take advantage of the underlying parallelized processing resources. This work focuses on the design of a fork and join parallel programming framework which can handle bulk data operations inherent in enterprise solutions. Techniques for designing parallel framework was followed and Amdahl's law is adopted to identify the appreciable speedup point so as to know the quantity of threads that could give optimum performance at any given time while exploiting available processor cores.**

***Index Terms*— Amdahl's law, Bulk data, Fork and join Parallelism, Programming model.**

I. INTRODUCTION

Surprisingly, most of the universe is inherently parallel. A great number of problems seeking solutions characteristically have abundance of natural parallelism upon which a software engineer can leverage. It is even possible to change the entire approach to solving a problem to utilize the inherent parallelism. Even most computer hardware today are built to provide parallelism capabilities. But generations of software architects have left performance to hardware and are only writing serial solutions to problems. When it comes to speed and turn-around-time, the common thinking pattern stops only at what the hardware and the system software can afford. We design our algorithms and write lines of code usually to solve parallel issues in a sequential manner forgetting that we live in a parallel world where solutions are better parallel aswell. Software engineering in itself involves "application of systematic, disciplined, quantifiable approaches to the development, operation and maintenance of efficient and effective software and the study of these approaches; that is, the application of engineering to software" [1]. But rather developed solutions are with the focus of "just working" not necessarily concerned with how well they work. Because the system hardware (specifically the processor) comes with better power over and over, generations of programmers / software architects have left performance to hardware. Leveraging on Gordon Moore's forecast that "the number of transistors incorporated in a chip would approximately double every twenty-four months" [2, 3] which translates to more processing speed. Software engineers then just rely on the speed available through the hardware. The reason for throwing in more transistors is actually to get more performance but this can only make systems faster to a limit. This is because it hit the wall on issues such as heat, power degeneracy, instruction-level parallelism, rate of clock, as well as chip scale. The new order is to consider having multiple cores or processors on a single chip which is referred to as the multi-core era. These processors are designed to run in parallel. Then automatically, the system assigns tasks to these processors as it deems fit to ensure they work in parallel in order to get anticipated performance [4]. For a method of scheduling to have a significant influence on performances, tasks are distributed over multiple cores [5]. But there is so much limitations in what the hardware can offer software developers if they leave the task to automatic parallelism of the hardware. This appears somewhat a level of irresponsibility implementing parallelizable solutions in the tradition serial manner. Sincerely, there is no magical hardware machine that will take serial code and turn it to parallel only because the processors now come in parallel. Therefore, the practical truce is that software developers should consider designing solutions in parallel to purposefully take advantage of the underlying hardware technology made available to them [6].

The principle in multi-core architecture is to keep number of cores constant for a number of years while increasing the number of threads in each core. A thread could be commonly described with a life cycle within the range of being created, started, running, waiting, and then terminated [7]. This implies keeping number of cores constant while increasing the degree of parallelism. To achieve full potential and performance of multi-core architecture, the software that will

IFERP connecting engineers... developing research

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Vol 11, Issue 1, January 2024

run on it must be written as parallel programs. This is because it is parallel program that will distribute computation among multiple cores thus enhancing throughput [8].

Many software frameworks have been developed to aid parallelism of software solutions such as Message Passing Interface, Java fork and join, MapReduce etc. In the study of [7] it was stated that comparison of parallel programming models would significantly provide useful insights for further work. Java fork and join could not handle bulk or big data operations. Thus, this study proposes a framework which extends Java fork and join to handle bulk data operations in parallel. This type of solutions is highly needed in enterprise solutions because of massive generation of data in their everyday operations.

Section 2 discusses literatures that reveal the trends in software processing, from serial programming to multi-processing and literatures on related works. Sections 3 to 5 are on the theoretical background to parallel design. Section 6 presents the design of the proposed framework for applying fork and join on big dataset.

II. LITERATURE REVIEW

Traditionally, computer software is written and known for serial computation. For a problem to be solved, usually an algorithm has to be designed and implemented as a sequence stream of both mathematical, logical and lexical instructions which are then executed by a system's central processing unit (CPU). Conversely, parallel computing makes use of several processing elements concurrently to resolve a given problem [9]. The whole process is reached by first dividing the whole problem into independent segments such that each processing division can execute each segment of the algorithm concurrently alongside with others. The processing elements can be varied and consist of resources like a single computer with several processors, networked computers, dedicated hardware, or their combination. The following sub-sections illustrated trends in software processing, advent of multiprocessor, successive pace of programming and implemented solutions:

A. Trends in Software Processing

The very first beginning was the days of ***serial processing*** and serial execution of programs on one processor. Traditionally, programs are design to run one instruction per time (one after the other) till completion. Serial execution of a program in-conjunction with numerous manual operations involved makes the execution slow and cumbersome. When programs are executed serially it causes the processor or the I/O devices to be idle at a time or the other during execution despite availability of scheduled job in the job stream. A way to solve this challenge is to allot extra or other jobs to the processor and I/O devices during the idling period. This interleaved execution of programs is called ***Multiprogramming*** [10]. ***Degree of Multiprogramming,*** that is, the number of programs actively competing for systems resources increases utilization. ***Multiprocessing*** means involving more than one processor in handling jobs or tasks or treads. In this turn, two or more processors are made to run multiple independent tasks concurrently in which communication and coordination between them are carefully managed. As an instance, MapReduce is multiprocessing under processor-level-parallelism. As performance improvement progressively becomes a critical concern, computer designers taught of enhancing chips speed by increasing their clock speed. Another direction most CPU designers progressed was parallelism. Two basic categories of parallelism include "***instruction-level parallelism*** and ***processor-level parallelism".*** The instruction-level looked into getting more instructions per second from individual instructions. The processor-level is how more than one CPU can work on the same problem [11].

Reference [12] presents a chart showing increasing transistor counts based on Moore's Law. However, a different method of applying them is in additional cores. First, single-thread performance has kept growing slightly, showing that it is still an essential quantity. These increases in performance were realized through careful power management and dynamic clock frequency adjustments. Thinking about the reality of the projected future, significant changes may not be experienced in frequency and power; nonetheless, more improvements in instructions per clock may somewhat raise single-threaded performance further, although the margin may not be that pronounced. What draws more interest are the transistor counts and quantity of cores. But to what length can Moore's Law be sustained? A likely estimate is that there can be an increase in the quantity of cores in relation to the proportion of the quantity of transistors. In recent times, Haswell Xeon CPUs already offer up to 18 cores, Knights Corner Xeon Phis is equipped with the 61 cores, therefore, one can submit that we are in an era where Amdahl's Law call for an application of parallel algorithms [13]. From an algorithmic standpoint, the quantity of cores at hand is not germane as knowing how to write massively parallel algorithms or make your solution take advantage of or efficiently utilize the hardware. One of the most popular plots in advancement in microprocessors and Moore's law is one named 35 Years of Microprocessor Trend Data by [12] with some extrapolation by C. Moore. Figure 1 shows the newer 42 Years of Microprocessor Trend Data.

ISSN (Online) 2394-2320
International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 11, Issue 1, January 2024

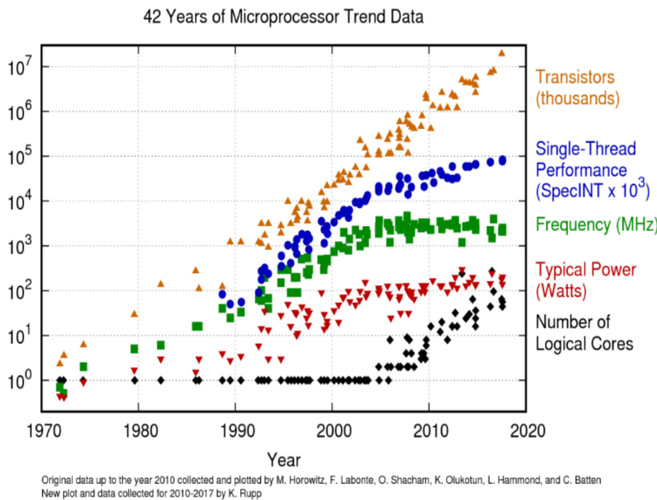

Figure 1: Forty-two years of microprocessor trend data [14]

The main reason for the introduction of parallel architectures were power constraints, as the electric power consumed by Central Processing Units and Graphic Processing Units (GPUs) (and thus the heat generated) increases in good approximation with the cubic power of clock frequency.

B. Parallel Computing Gap

Parallelism means that an application splits its tasks up into smaller subtasks which can be processed in parallel, i.e., on multiple CPUs at a go. Parallelism could be achieved on distributed systems, multi-processor or multi-core. If there is only one processor, then there can be concurrent execution but not parallel. Concurrency implies that an application is working on more than one tasks at the same time [15]. A well-written concurrent program might run efficiently in parallel on a multiprocessor.

Parallel computing is "the use of two or more processors (cores, computers) in combination or simultaneously to solve a single problem" [16]. Substantial percentage of the universe is naturally parallel; many unsolved problems have equally many natural parallelisms. Sometimes, the entire approach to solving an identified problem may be changed just to make use of the natural parallelism. For instance, a parallel computer of about 50 processors does not make a program run 50 times faster. A good factor is that the sequential program might probably not have been very good [17]. This may not be generally obvious but many of the parallel processors are habitually idle. Traditionally, experiences of most programmers in the area of parallel computing are little; this is coupled with the fact that there are very little parallel programs to emulate. Parallel program developers are equally behind on their learning curves and also existing materials made for serial systems are reused, even when such materials cause performance difficulties [18].

Majority of serial computers have the same fundamental arrangement, but this is not the same for parallel computers. Parallel computers like the popular cluster systems are basically just an assemblage of computers interconnected together with Ethernet. They make use of simple instructions which are similar to read and write to be able to communicate among the processors [16]. The most popular method for doing this communication is the Message Passing Interface (MPI). Parallel computers of this like are described as message-passing systems, or distributed memory computers. To bring the concept of parallel computing or programming to a simple illustration, if there is a job of painting a building to be done, and the building contains 5 rooms, and the painting of one room is independent of the other and also will require the same amount of time to get each room painted. One can assign the task of painting each room in the building to five different painters (workers) and consequently get the entire job done about 5 times faster. Tasks that are easily parallelizable are sometimes called ***embarrassingly parallel.*** An example is a brute-force search. We may not have to long for 100% embarrassingly parallel situation which may not always happen because it is difficult to parallelize a lot of real-world computational problems excellently without sustaining high cost of inter-processor communication and coordination.

In the painting task example, we could get five-fold speed-up compare to a single painter having to do the entire job. But complication sets in if one of the rooms is twice the size of the others, then the 5 painters will not achieve a 5-fold speedup because the overall completion time is dominated by the one room that takes the most time. Hence, Amdahl's law formula is needed which captures the extent to which any complex job can be sped up which is restricted by how much of the job should run serially.

Amdahl defines the "speedup, S, of a job as the ratio between the times it takes one processor to complete the job versus the time it takes n number of concurrent processors to complete the same job. Amdahl's law characterizes the maximum speedup S that can be achieved by n processors collaborating on an application, such that p is the fraction of the job that can be executed in parallel."

A program or an algorithm that is a candidate for parallelization can be split into two parts:

1) A part which can be parallelized- parallelizable
2) A part which cannot be parallelized -non-parallelizable

According to this law, if it takes standardized time 1 for a processor to complete the job, with n simultaneous processors, the parallel part takes time p/n and the serial part takes time 1-p, then the overall parallelized calculation takes:

$1 - p + p/n$

Then Amdahl's law [**19, 20**] implies that:

The speedup, i.e., the ratio between the sequential and parallel time is given in Equation 1 as:

$$S = \frac{1}{1-p+p/n} \quad (1)$$

IFERP connecting engineers...developing research

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Vol 11, Issue 1, January 2024

Following the earlier example, assume that each room is 1 unit, and the single large room has two units. Assigning one painter (representing processor) per room means that 5 of 6 units can be painted in parallel, meaning that p = 5/6,

and $1 - p = 1 - 5/6 = \frac{(6-5)}{6} = 1/6.$

$$p/n = 5/6/n = 5/6/5 = \frac{5}{6*5} = 1/6$$

Therefore, the resulting speedup S becomes:

$$S = \frac{1}{1/6 + 1/6} = 3$$

In summary, 5 painters worked on 5 rooms where one room is twice the size of the other yields only a 3-fold speed up. Invariably, this result is affected by the non-parallelizable part of the task. And this can reduce further down as the number of tasks increases involving non- parallelizables [21]. This is because the shared part of the job will require careful coordination. One may only need to seek the point where the speedup is appreciable and when it is no longer necessary.

Previously, when bigger problems are to be solved, owners of these problems could often wait a little for a faster computer to emerge. This was mainly due to *Moore's law*, which majority took to mean that computer's speed would double their current speed about every two years. Though things are not going that way any longer; if the GigaHertZ (GHz) speed of the processors in a desktop computer is considered, say 2 years ago, against the speed now, the difference is not outstanding. It is getting impracticable to make reliably low-cost processors that run significantly faster. The point of deflection that promotes doubling up is the quantity of processors on a particular chip. Today, it is practically difficult to get a computer that is not parallel. Even as small as an Apple watch is, it has 2 cores. Some notable graphics processing chips have 128 specialized compute cores, and some supercomputers incorporate these chips (although the chips are somehow challenging to use efficiently). The Tianhe-2 supercomputer has > 3,000,000 cores and a theoretical peak performance of > 50 Petaflops, i.e., 50,000,000,000,000,000 floating point operations per second [22]. The amount of cores/chip will increase constantly, and hence parallel computing is essential to use the available hardware [23]. This is particularly true for tasks that are computation-intensive such as simulations, big data analytics, complicated systems optimization etc. It may be sufficient to say that Parallel computers are quite easy to develop; the greater part of the effort is in software design. Some smart-phones now have 8 cores (processors) and even more. These are referred to as multi-core chips. Also, there are graphics processing units with more than 100 highly specialized processors. This is near to what Moore postulated when he said that the number of transistors would keep doubling. The rate of expansion will slow down, but significant increases in the amount of transistors might still continue. If a single processor cannot offer the needed speed because of its inherent limitation, considering Amdahl at a reasonable level on multicores should offer a better advantage if software professionals at large would closely consider parallelizing their solutions.

By theoretical calculation, Amdahl's law shows that the parallelizable part can be executed faster by adding more hardware to it. This is not completely true because it gets to a point where increase in n (number of CPUs) will make no more difference. The other option is that the non-parallelizable part could be executed faster by optimizing the code. The algorithm might even be modified to ensure the non-parallelizable part is generally reduced, by making some of the task to become parallelizable (as much as possible). While Amdahl's law promotes calculating the theoretical speedup of parallelization of an algorithm [24], other factors such as speed, CPU cache memory, and network cards may also be checked to get the totality about overall speed.

C. Designing Parallel Programs

Parallel programming is the name generally given to the methods that take advantage of computers with two or more processors (multicore). It is essential to determine which designing technique is appropriate for a parallel algorithmic problem. Notable design techniques include Divide and Conquer, Dynamic Programming, Linear Programming, Greedy Method, Backtracking, Branch and Bound.

To be able to design parallel programs, the followings are important:

1) *Problem characteristics*

i) Understand the Problem and the Program

Sometimes, it might be a problem that you need to understand, at other times you might need to understand existing serial codes. This depends on where you are starting from i.e. whether there is an existing solution or you are providing a fresh one.

ii) Identifying dependencies and independencies in the Problem

To provide a fresh solution to a problem, what comes first and most paramount is to understand the problem to solve in parallel. Then one must find out if the problem at hand is a type that can truly be parallelized. An example of a problem with little or no parallelism is the one displaying high dependency, either code or data e.g: Computation of F, the Fibonacci series (0,1,1,2,3,5,8,...) using the formula in the following Equation 2:

$$F(n) = F(n-1) + F(n-2) \quad (2)$$

Obviously, the computation of the F(n) value uses those of both F(n-1) and F(n-2), which must be computed first.

IFERP connecting engineers...developing research

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Vol 11, Issue 1, January 2024

iii) Identifying hotspots sections in the code

When dealing with an existing serial code, we need to first determine the part of the code where major work is done or using computer resources mostly. This is done through asymptotic analysis of the algorithm. Those code parts using up computer resources like CPU and memory is referred to as hotspots. Then the hotspots should be parallelized if they are independent parts [25].

iv) Identify slow areas of code

Identify areas that are predominantly slow and thus become bottlenecks due to I/O. This can be handled by either restructuring the algorithm or use a new algorithm or remove the predominantly slow sections.

v) Consider reusability

Consider reusing an existing third-party parallel software or framework and use of libraries that can automatically be used for developing parallel software.

2) Partitioning

Partitioning of a problem has to do with decomposition or breaking down of problem or jobs into manageable units called chunks. A chunk can be 64bits or 128bits. One of the important first steps in designing a parallel program is to break the problem into smaller discrete chunks. The two methods of doing this are domain decomposition and functional decomposition.

(i) Domain Decomposition

In domain decomposition, the problem is viewed from the sides of its associated data as shown in Figure 2 below. Thus, its dataset is broken into pieces or chunks that are manageable. Each manageable chunk is called a task. Each chunk is managed in separate processors where software works on each task independently and simultaneously i.e., in parallel. Solution from each task is later merged to provide the main solution.

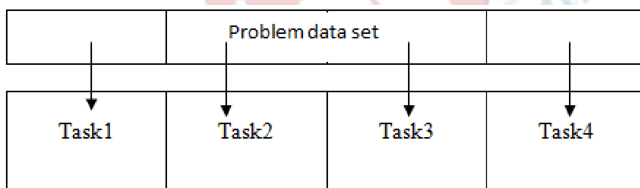

Figure 2: Data partitioning

(ii) Functional Decomposition

In functional decomposition, computation rather than data is decomposed into subparts, where each part performs a specific role. Thus, each subpart or task works to perform a portion of the overall work to be done as illustrated in the following Figure 3:

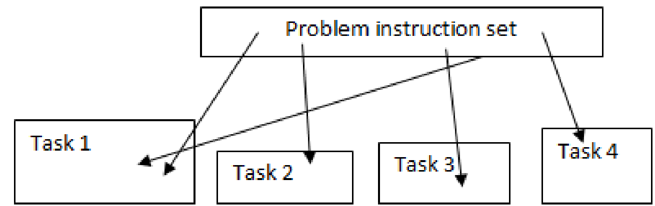

Figure 3: Functional decomposition of computation split into different tasks

3) Synchronous vs. asynchronous communications in Parallel Program

Sharing of data between tasks involves communications. Communication can be either synchronous or asynchronous. A Synchronous communication is the one where there is handshaking between tasks sharing data. Implementation of synchronous communication can be done explicitly by programmers into the parallel code. This increases coding workload on the programmer because the programmer needs to worry about providing solution to the problem at hand and as well handle the synchronous communication. In other cases, the implementation is done implicitly. In this case, synchronous communication code is integrated at low level without programmer's knowledge. In synchronous communications, job needs to wait for the completion of other job communication, this is known as blocking communication. Asynchronous communications is a non-blocking communication which allows other jobs waiting for communication to do other useful tasks i.e. they do not wait idling.

D. Closely Related Works on Parallel Programming Models

This section reviewed literatures on designing of parallel programs to building tools for converting serial programs to parallel ones using shared-memory architecture.

Reference [26] developed two models; the first one is a computational model for designing and analyzing algorithms that runs on shared memory multi-core architecture, and the second one is Software and Algorithm for running on Multi-core (SWARM). The computational multi-core model considered the issues of number of cores integrated on a single chip, memory bandwidth usage, memory cache and synchronization. These problems were resolved by the computational model with the use of cache-aware approach. SWARM is an open source library which contains parallel programming framework for developing algorithm that works on multi-core systems.

Reference [27] introduced Ateji® PX1 a Java preprocessor which converts serial programming into parallel programs which can efficiently use the underlining multi-core chip. Ateji® PX1 attempts to make parallel programs work efficiently on shared-memory architecture and overcomes the pitfall encountered when developers individually developed a parallel program. Such pitfalls are race condition, deadlock,

IFERP
ISSN (Online) 2394-2320
International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 11, Issue 1, January 2024

late time to market, complex and error-prone parallel programs. By converting serial programs to parallel ones Ateji® PX1 could enhance central processing unit (CPU) to accommodate more jobs at the same time on multiple cores. This implies reduction in processing time, full and efficient utilization of all the cores.

Reference [28] in their research compared two parallel frameworks; Java fork/join and MapReduce with respect to scalability and programmability. In terms of scalability, Java fork/join cannot scale well with large inputs because of its shared-memory architecture while MapReduce scales well on very large (big dataset) input size with its shared-nothing architecture. In terms of programmability, MapReduce has fewer lines of code than Java fork/join. The authors further compared MapReduce and Java fork/join which have parallel construct with sequential C and Haskell sequential which are without parallel construct. It was found that sequential C has 288 lines of code, while Haskell has 66 lines of code but both could not scale well with large input sizes.

The work of [29] established that high performance computing requires parallel processing. Their results show jobs are done more easily and time is also saved when load is shared via efficient multicore utilization. The major reasons for much time consumption are performance measure on huge data and also algorithms operating on this data. The study touched on extension to mobile applications which are rapidly under development in many applications.

Reference [30] considers creating active knowledge technology as potential features of parallel programming systems. Worthy of consideration in generating parallel programs is the properties of model's data because the data properties commonly define the numerical model's behavior. The study also described the development of intelligent algorithms for generation of parallel programs of numerical supercomputer simulation as it affects the behavior of models.

In the overview carried out by [31], it was stated that study in the field of Internet of things (IoT) have demonstrated the possibility of producing large volume of data and computation among various devices of the IoT. Then the Industrial Internet of Health Things (IIoHT) is an extension of Internet of Health Things (IHoT). The IoHT with diversity of tasks such as observing, consulting, monitoring, and treatment process of remote exchange data processes would benefit greatly from parallel computing and influence IIoHT.

III. PROPOSED FRAMEWORK FOR FORK AND JOIN PARALLELISM IN BULK DATA

The Fork and Join model simply break (forks) a task into subtasks in a way that each subtask is independently handled by a resource such as the processor and after each subtask's operation has performed expected execution, they can be joined back to form a coherent solution which serves as the result of the experiment. This disintegration is applied recursively until sub-problems are so small that sequential solution becomes faster. The techniques of operation here is obviously the Divide-and-conquer method. Naturally, a divide-and-conquer breaks a whole problem down into chunks or sizes that can be easily solved individually. These chunks are called sub-problems, each individual chunk is solved to provide a solution, and their several solutions are combined to form a singular solution which is for the whole problem as illustrated in the following Figure 4:

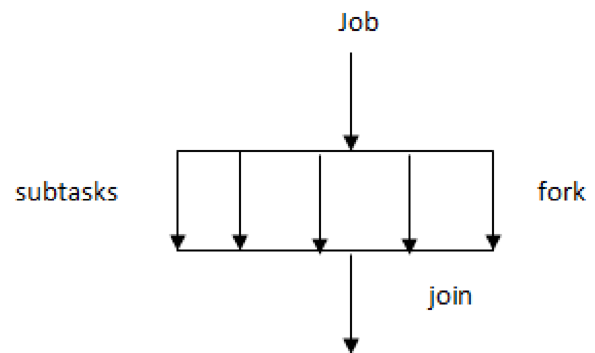

Figure 4: Illustrating fork and join technique

The following Figure 5 illustrates the proposed architectural model for Fork and Join Parallelism as applied to Bulk Data implementation. This presents how a pool of massive data can be modeled to make an optimal use of the underlying multi-core processor hardware rather than assuming the solution will run in parallel because the hardware is by default in parallel. Millions of records are loaded onto a memory list, while the records are divided into threads. In the process, Amdahl laws is adopted to compute the optimal speedup for the said process in execution which helps in partitioning tasks into their parallelizable number and therefore distributed to the multi-core hardware to ascertain an optimal utilization of the existing hardware. These subtasks are executed by the multi-core processors and results are delivered in parallel. Work stealing can also take place in the midst of the multithreaded programs to make the parallel computing effective. This is a way of playing two sets of idle and saturated processors against each other. Work stealing distributes the scheduling work over idle processors in order to ensure that all processors have tasks to do and as such no scheduling overhead occurs. This is done efficiently in terms of execution time, memory usage, and inter-processor communication. This is what is employed in the scheduler for the Java fork/join framework.

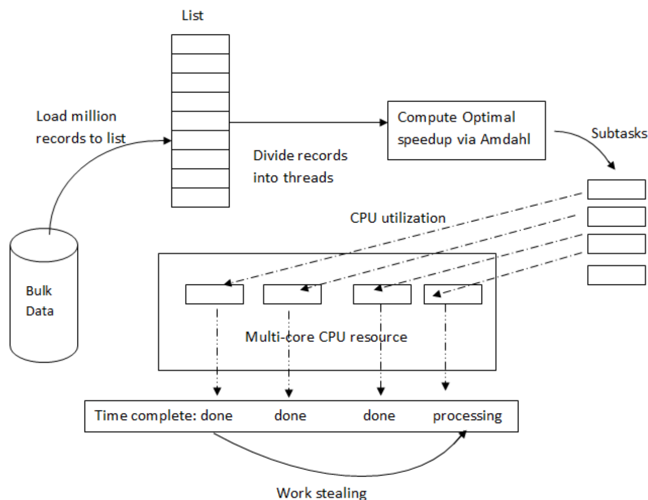

Figure 5: Proposed Architectural Framework for Fork and Join Parallel Programming Model in Bulk Data

It is important to set a reasonable sequential threshold which will also aid division of tasks. Also, making tasks smaller enhances parallelism, increases load balancing and improves throughput.

IV. ECONOMIC IMPORTANCE OF PARALLELISM

Successful parallelism provides improved performance; better efficiency; more productivity; reduces time to market; and maximized profits because it is designed towards customer specifications and business goals. Furthermore, parallel programs are easily modified as business requirements changes. With a well programmed parallel computing, CPU resources is efficiently utilized because all available cores will be engaged such that computer can handle more different jobs at the same time. Parallel computing can better represent real world events than serial programming making parallelism the future of computing. In addition, to make efficient use of modern computer architecture which uses parallel hardware, software that runs on it should be parallel software. Parallelization is also significant because it reduces processing time, increases throughput which is easily noticeable when dealing with big data operations.

V. FUTURE RECOMMENDATION AND CONCLUSION

With the perceived future of smart systems and Artificial Intelligence, processing speed is a key requirement. Current turn of events clearly indicate that the software professionals need to think more in the direction of parallelizing their inventions just as most of their hardware counterparts are busy doing. Many problems already present parallel ways of solving them. It is even possible to change the entire approach to solving a problem in order to take advantage of the inbuilt parallelism in the problem. Though parallel programs are not simple to write but if careful attention is given to this, the outcome could be incredibly great [32]. In the generation where there is now incredibly massive bulk data to deal with, software developers need to seek algorithms and techniques that will directly take advantage of the underlying parallelized processing resources.

REFERENCES

[1] IEEE, (1990).Institute of Electrical and Electronics Engineers, IEEE 610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology. Published Date:1990-12-31, Reaffirmed:2002-09-12.

[2] G. Moore, “Cramming more components into integrated circuits,” *Electronics Magazine*, vol. 38, no. 8, pp. 114–117, 1965.

[3] M. T. Heath, “A tale of two laws,” *The International Journal of High Performance Computing Applications*, vol. 29, no. 3, 320–330, 2015.

[4] C. Saidu, A. A. Obiniyi and P. Ogedebe, "Overview of Trends Leading to Parallel Computing and Parallel Programming," *British Journal of Mathematics & Computer Science,* vol. 7, no. 1, pp. 40-57, 2015.

[5] A. Klilou and A. Arsalane, "Parallel implementation of pulse compression method on a multi-core digital signal processor," *International Journal of Electrical and Computer Engineering (IJECE),* vol. 10, no. 6, pp. 6541-6548, 2020.

[6] G. E. Blelloch and B. M. Maggs, “Parallel algorithms,” Algorithms and theory of computation handbook: special topics and techniques (2nd ed.). Chapman & Hall/CRC, 2010.

[7] M. N. Jamaluddin, A. Ismail, A. Rashid and T. T. Takleh, “Performance Comparison of Java based Parallel Programming Models,” *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 16, no. 3, pp. 1577-1583, 2019.

[8] B. Shekhar and C. A. Andrew, “The future of Microprocessors,” *Communications of ACM*, 67-77, 2011.

[9] Y. Ben-Asher, “Basic Concepts in Parallel Algorithms and Parallel Programming. Multicore Programming Using the ParC Language,” Undergraduate Topics in Computer Science. Springer, London, 2012.

[10] M. Milenkovic, “Operating Systems, Concepts and Design,” (2nd e.d), Tata McGraw-Hill Companies, Inc., New York, 2005.

[11] A. S. Tanenbaum, “Structured Computer Organization,” (6th ed.), Upper Saddle River, New Jersey, Pearson Prentice Hall, 2012.

[12] M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten, “35 *Years of Microprocessor Trend*,” 2010. Available: https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

[13] I. Cutress, and A. Shilov, "The Larrabee Chapter Closes: Intel's Final Xeon Phi Processors Now in EOL, " 2019. Available: https://www.anandtech.com/show/14305/intel-xeon-phi-knights-mill-now-eol. Retrieved June 10, 2021.

[14] C. Rupp, “42 Years of Microprocessor Trend Data,” 2018. Available: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

[15] B. Hemprasad, “Concurrency Vs Parallelism”.Shri Guru GobindSinghji Institute of Engineering and Technology, 2013.

[16] Q. F. Stout and R. Miller, “Algorithms Techniques for regular networks of processors, ”Algorithms and Theory of Computation Handbook, (2nd ed.), M. Atallah, ed., 46:1–18, 2009.

[17] Network World, Published by IDG Network World Inc. 18(9), 108 pgs, February 6, 2001, ISSN 0887-7661

[18] *A. C. Andrew, “Parallelism for the Masses: Opportunities and Challenges,” Intel Corporation, Carnegie Mellon University, 2008.*

[19] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities". *Proceeding AFIPS '67* (Spring) Proceedings of the April 18–20, 1967, Spring Joint Computer Conference: 483–485.

[20] M. McCool, J. Reinders and A. Robison, “Structured Parallel Programming: Patterns for Efficient Computation, “ *Elsevier*, p. 61, 2013.

[21] H. Maurice and N. Shavit, "The Art of multiprocessor programming,"

IFERP connecting engineers... developing research

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Vol 11, Issue 1, January 2024

(1st ed.). USA: Morgan Kaufmann Publisher: Elsevier imprint, 2008.

[22] D. Alba, "China's Tianhe-2 Caps Top 10 Supercomputers". *IEEE Spectrum*, 2013.

[23] M. T. Heath and E. Solomonik, "Parallel Numerical Algorithms," Lecture Notes: Department of Computer Science, University of Illinois, Urbana-Champaign, 2019. Available: https://solomonik.cs.illinois.edu/teaching/cs554_fall2017/notes/chapter_01.pdf

[24] C. A. Navarro, N. Hitschfeld and L. Mateu, "A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures," *Communications in Computational Physics*, vol. 15, pp. 285-329, 2013.

[25] B. Barney, "Introduction to Parallel Computing Tutorial," Livermore Computing Center, California, United States, 2021. Available: https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial

[26] D. A. Bader, V. Kanade and K. Madduri, "SWARM: A Parallel Programming Framework for Multicore Processors," Georgia Institute of Technology, College of Computing, 2007. Available: http://multicore-swarm.sourceforge.net/#documentation

[27] M. Giri and R. Rahul, "Leveraging Multi-core Processors Through Parallel Programming cognizant 20-20 insights", Teaneck, New Jersey (U.S.), 2011.

[28] R. Stewart, and J. Singer, "Comparing Fork / Join and MapReduce. Mathematical and Computer Sciences, Heriot Watt University, 2012.

[29] K. Sujatha, P. V. Rao, A. A. Rao, V. G. Sastry, V. Praneeta, and R. K. Bharat, "Multicore parallel processing concepts for effective sorting and searching," *International Conference on Signal Processing and Communication Engineering Systems, 2015,* pp. 162-166, doi: 10.1109/SPACES.2015.7058238.

[30] V. Malyshkin, "Parallel computing technologies 2018: Automatic parallel implementation of applications." *The Journal of Supercomputing,* vol. 75, 7747–7749. 2019.

[31] X. Yang, S. Nazir, H. Khan, M. Shafiq, and N. Mukhtar, "Parallel Computing for Efficient and Intelligent Industrial Internet of Health Things: An Overview," *Complexity*, vol. 2, pp. 1-11, 2021.

[32] I. Öz and S. Arslan, "Predicting the Soft Error Vulnerability of Parallel Applications Using Machine Learning," *International Journal of Parallel Programming*, 3, 2021.

IFERP connecting engineers...developing research