THE IMPROVEMENT OF PARALLEL SCIENTIFIC COMPUTING BY THE APPLICATION OF COMPLEX ALGORITHM

Ebole Alpha F[1], (b) Awodele O[2]. and (c) Kuyoro S.O[3]

alphaechoe@yahoo.com delealways@yahoo.com; afolashadeng@gmail.com

1. Computer Science Department, School Of Technology, Lagos State Polytechnic, Ikorodu Lagos, Lagos State, Nigeria.

[2,3] Computer Science Department,School of Computing and Engineering Sciences, Babcock University, Ilishan Remo, Ogun State, Nigeria.

ABSTRACT.

It is now very clear that silicon based processor chips are reaching their physical limits in processing speed, as they are constrained by the speed of electricity, light, and certain thermodynamic law. A viable solution to overcome this limitation is to connect multiple processors working in coordination with each other as well as using complex Algorithm to solve grand challenge problems. Hence, high performance computing requires the use of massively parallel processing (MPP) systems containing thousands of powerful central processing unit and complex Algorithm like the Divide and conquers algorithms. It involve breaking up a problem into several smaller instances of the same problem, solve these smaller instances by assigning different processors to the smaller instance of the same problem and then combine the solutions into a solution to the original problem by a central processor. Naturally, for this approach to work, the smaller instances should be simpler than the original problem, and combining their solutions together should be easier to do than solving the original problem.

INTRODUCTION

The feasibility of parallel processing can be demonstrated by the application of neurons in the brain. Aggregate speed with which complex calculations carried out by neurons is tremendously high, even through individual response of neurons is too slow (in terms of milli seconds) . Since the invention of the first transistor in 1958 an empirical "law" has continued to predict our ability to manufacture in ever smaller dimensions. Gordon Moore, co-founder of Intel, made the observation that approximately every 18 months the number of transistors that could be mass produced in a given area doubled. From 1958 until about 2002 this statement was equivalent to saying that the speed of the processor also doubled every 18 months. Since then the increase in performance of a single core has increased much more slowly. The impact of this decrease in performance growth rate and its consequence for scientific computing are the main motivating force behind this Journal. The solution of the hardware designers to the inability to significantly increase single-threaded performance was to increase the explicit parallelism both in the hardware and in the programming model. No longer can software be written in a sequential fashion relying on advances in hardware to improve performance. Software must now be written to take advantage of the parallelism inherent in the processors by explicitly expressing the parallelism of the algorithms. The parallel computing era start with a development in the Hardware architectures, followed by system software (compiler, and operating system), applications and reaching its saturation point with its growth in problem solving environments. Parallel computing involved the processing of multiple tasks simultaneously on multiple processors by dividing the task into subtasks by divide and conquer technique in other to improve parallel scientific computing in the area of Computational requirement , Sequential architecture , Hardware improvement, Vector processing and Network technology.

REVIEW OF RELATED LITERATURE

Different types of high performance machines based on the integration of many processors are available. Some of the most important are vector multiprocessors (a small number of high performance vector processors); massively parallel processors (hundreds to millions of processors connected together with shared or distributed memory); and networked computers (workstations connected by a medium or high speed network, working as a high performance virtual parallel machine) as stated by (Lau 2004). (Foster and Kumar et al 1998) look at how Parallelism can leads to the need to design new algorithms, specifically designed to take advantage of concurrency, because many times the best parallel solution strategy cannot be achieved by just adapting a sequential algorithm. (Chapman and Wolf et al 2005). Programs implementing these algorithms are developed and executed in parallel computing environments. (Blackford et al, 2000) explain that the software consists of parallel programming tools, performance tools and debuggers associated to them, and some libraries developed to help in solving specific classes of problems. (Geist et al 2004) PVM (Parallel Virtual Machine) which is a widely used message passing library, created to support the development of distributed and parallel programs executed on a set of interconnected heterogeneous machines. The book Efficient Parallel Algorithms by (Gibbons and Rytter 2005) has a brief discussion of parallel models of computation followed by substantial material on parallel algorithms. The large work Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes by (Leighton 2004) contains a detailed discussion of many different types of parallel models and algorithms for them. Synthesis of Parallel Algorithms, edited by (Reif 2004), contains twenty chapters organized around parallel algorithms for particular types of problems, together with an introductory chapter on P- completeness (Greenlaw

2005). The chapter Parallel Algorithms for Shared-memory Machines by (Karp and Ramachandran 2004] in the Handbook of Theoretical Computer Science describes a variety of highly parallel algorithms for shared memory machines. In the same handbook, the chapter A Catalog of Complexity Classes by Johnson is a thorough overview of basic complexity theory and of the current state of knowledge about most complexity classes. It is an excellent reference for establishing the context of each class and its established relationships to others.

HARDWARE ARCHITECTURE OF INSTRUCTION EXECUTION

(i) **Single Instruction on Single Data** (SISD): it is a uniprocessor machine that operates in sequential form. The processing speed depends on the rate at which computer can transfer information internally.

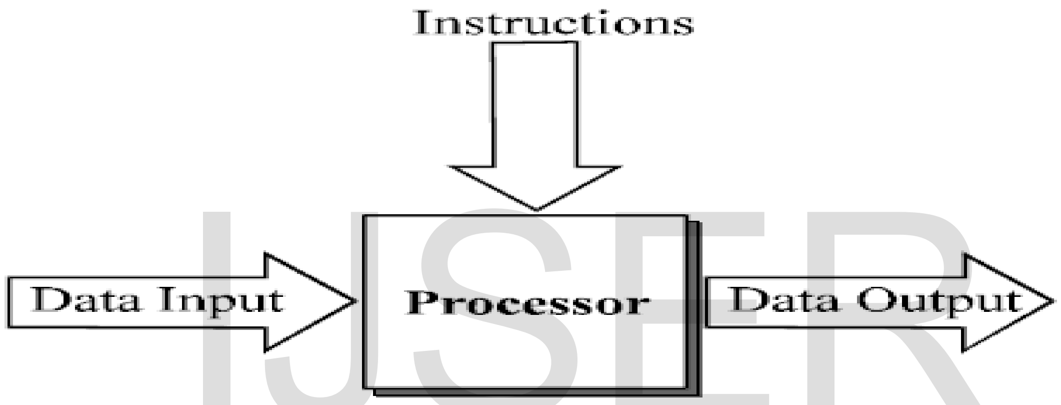


Fig 1: Single Instruction on Single Data Instruction Architecture.

(ii) **Single Instruction on multiple Data** (SIMD): It is a multiprocessor machine capable of executing the same instruction on all the CPU'S but operating on different data. It also suited for scientific computing and involved a lots of vector and matrix operations.

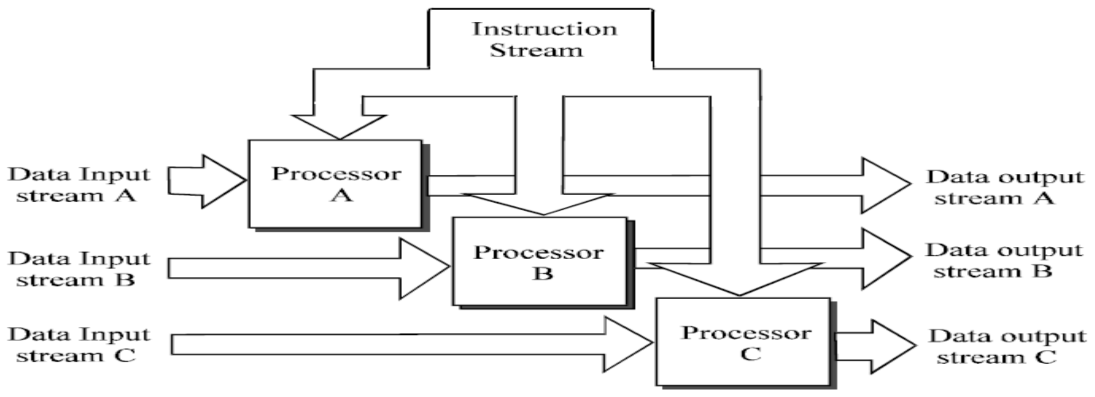


Fig 2: Single Instruction on multipleS Data instruction Architecture.

(iii) **Multiple Instructions on Single Data**. (MISD): It is a multiprocessor capable of executing the different instructions on different PEs, but all of them operating on the same data – set.

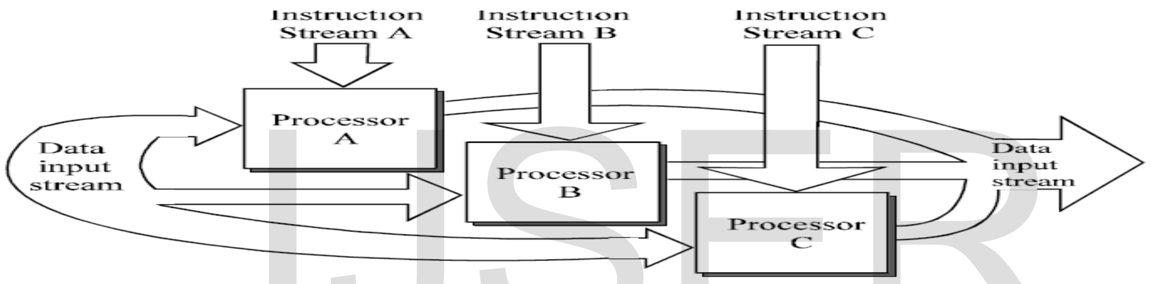


Fig 3. **Multiple Instructions on Single Data Architecture**

(iv) **Multiple instructions on Multiple Data** (MIMD): It is a multiprocessor machine capable of executing multiple instructions on multiple data sets. The PEs in MIMD machines work asynchronously.

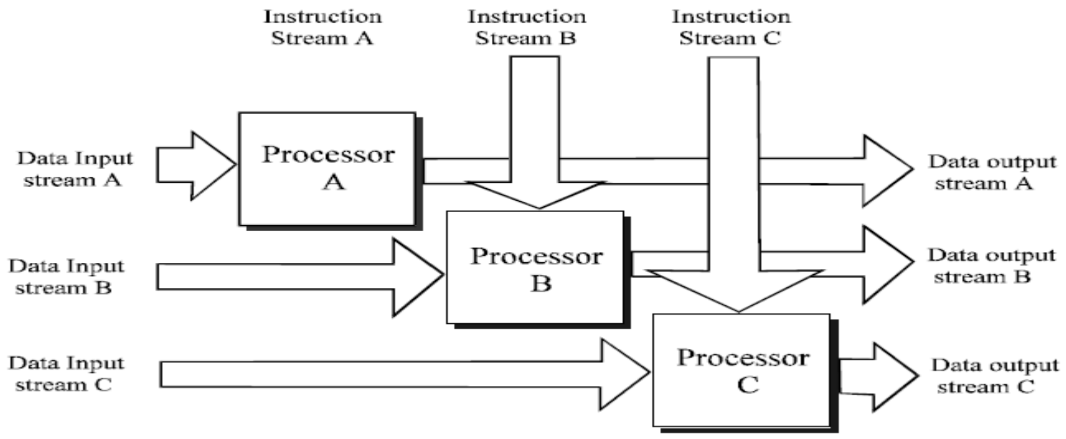


Fig 4. Multiple instructions on Multiple Data Architecture

MIMD Machine are categories into two:

Shared memory MIMD Machine (tightly – coupled multiprocessor). And Distributed memory MIMD Machine (loosely–coupled)

The Shared memory Machine: In this model, all the PEs is connected and has access to a single global memory. The modification of the data store in the global memory PEs is visible to all other PEs. Examples are silicon graphics machines, sun's symmetric multi- processing. And their characteristics are Manufacturability, Programmability Un- reliability, Un – scalability that is memory contention

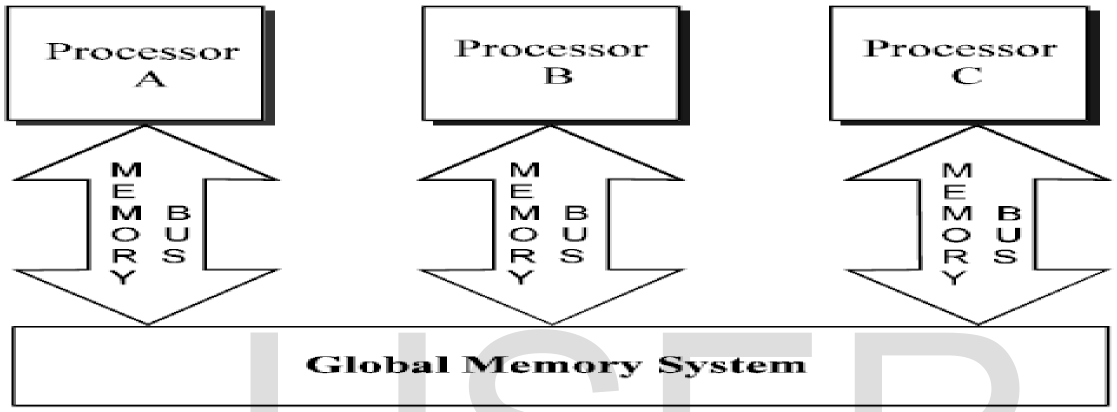


Fig 5. **Shared memory Machine MIMD Architecture**

Distributed memory MIMD Machine: In this model, all the PEs has its own local memory. The communication between PEs in this model takes place through the interconnection network. And this connection network can be configured to tree, mesh, cube, etc. Example are C-DAC's PARAM, IBM's SP/2, Intel's Paragaon, etc. and its characteristics are Manufacturability, Unprogrammability, Reliability, Scalability.

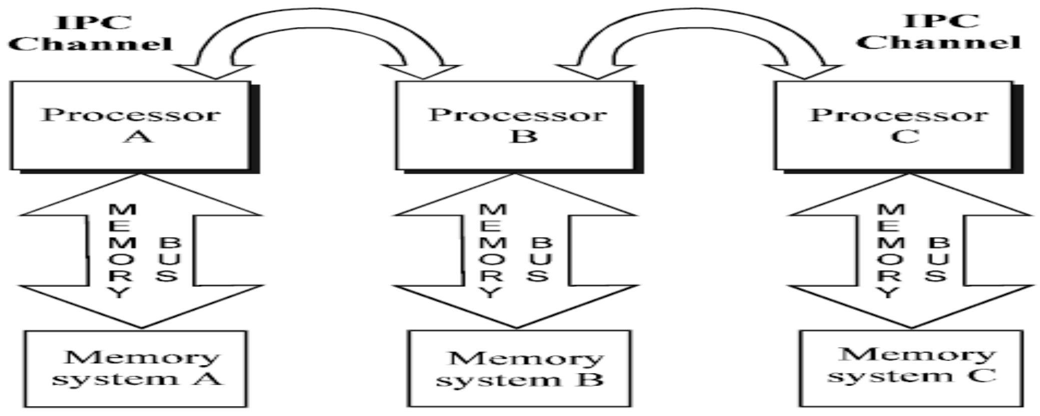


Fig 6. **Distributed memory MIMD Machine MIMD Architecture**

APPROACHES TO PARALLEL PROGRAMMING

A sequential program is one which runs on a single processor and has a single line of control. To make many processors collectively work on a single program, the program must be divided into smaller independent chunks so that each processor can work on separate chunks of the problem.

The approaches are:

Data parallelism: it uses divide and conquer technique.

Process parallelism: A give operation has multiple activities which can be processed on multiple processors.

Farmer and worker model: it involved job distribution, where the farmer is the master and the workers are the slaves.

Parallelism

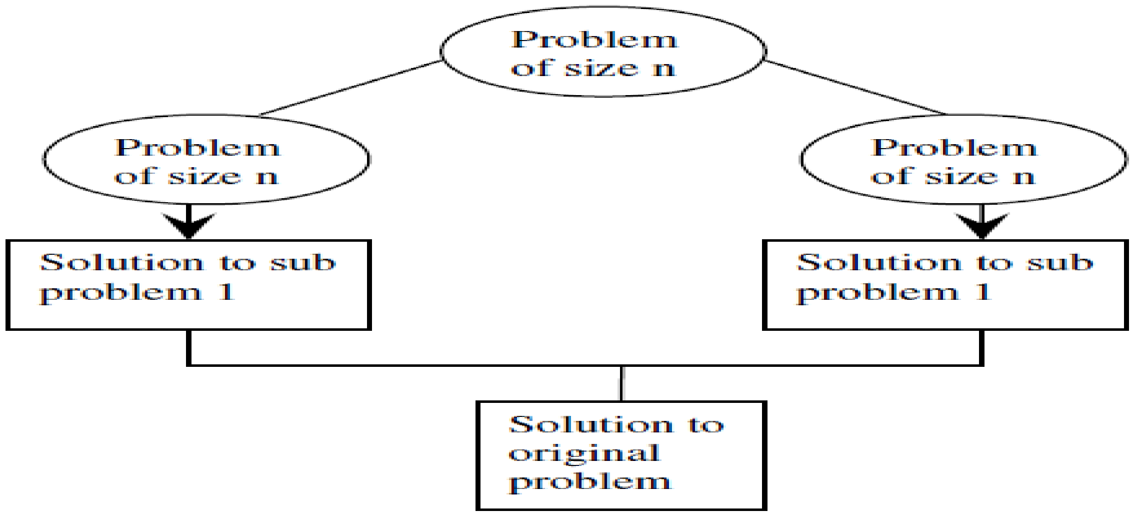
For many years, we could count on processor clock speeds increasing at a steady rate. Physical limitations present a fundamental roadblock to ever-increasing clock speeds, however: because power density increases super linearly with clock speed, chips run the risk of melting once their clock speeds become high enough. In order to perform more computations per second, therefore, chips are being designed to contain not just one but several processing "cores." We can liken these multicore computers to several sequential computers on a single chip; in other words, they are a type of "parallel computer." In order to elicit the best performance from multicore computers, we need to design algorithms with parallelism in mind. This model has advantages from a theoretical standpoint, and it forms the basis of several successful computer programs, including a championship chess program.

DIVIDE & CONQUER.

Divide & conquer is a general algorithm design strategy with a general plan as follows:

1. DIVIDE: A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
2. RECUR: Solve the sub-problem recursively.
3. CONQUER: If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.

Diagram 1 shows the general divide & conquer plan


Problem of size n
Problem of size n
Problem of size n
Solution to sub problem 1
Solution to sub problem 1
Solution to original problem

The base case for the recursion is sub-problem of constant size. And the benefits are

(i) For solving conceptually difficult problems like Tower Of Hanoi.

(ii) Results in efficient algorithms

(iii) Divide & Conquer algorithms are adapted for execution in multi-processor machines

(iv) Results in algorithms that use memory cache efficiently.

When the sub-problems are large enough to solve recursively, we call that the ***recursive case***. Once the sub-problems become small enough that we no longer recurse, we say that the recursion "bottoms out" and that we have gotten down to the ***base case***. Sometimes, in addition to sub-problems that are smaller instances of the same problem, we have to solve sub-problems that are not quite the same as the original problem. We consider solving such sub-problems as part of the combine step. The first one solves the maximum sub-array problem: it takes as input an array of numbers, and it determines the contiguous sub-array whose values have the greatest sum. Then we shall see two divide-and-conquer algorithms for multiplying $n \times n$ matrices. One runs $\Theta(n^3)$ in time, which is no better than the straightforward method of multiplying square matrices. But the other, Strassen's algorithm, runs in $O(n^{2.81})$ time, which beats the straightforward method asymptotically.

CONCLUSSION AND RECOMMENDATION

Unfortunately, programming a shared-memory parallel computer directly using static threads is difficult and error-prone. One reason is that dynamically partitioning the work among the threads so that each thread receives approximately the same load turns out to be a complicated undertaking. For any but the simplest of applications, the programmer must use complex communication protocols to implement a scheduler to load-balance the work.

This state of affairs has led toward the creation of ***concurrency platforms***, which provide a layer of software that coordinates, schedules, and manages the parallel-computing resources. Some concurrency platforms are built as runtime libraries, but others provide full-fledged parallel languages with compiler and runtime support.

To amplifier the quick execution of programs, we can use the complex Algorithm in terms of designing faster Algorithm in line with Farmer and Workers Model. The running time of the algorithm is the sum of running times for each statement executed; a statement that takes $\mathbf{c_i}$ steps to execute and executes **n** times will contribute $\mathbf{c_{in}}$ to the total running time. For the best case: which is a linear function. an + b

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

We can express this worst-case running time as an₂ C b**n** C c for constants a, b, and c that again depend on the statement costs $\mathbf{c_i}$; it is thus a ***quadratic function*** of **n.**

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

REFENRENCES

1.Aiex, R.M., S.L. Martins, C.C. Ribeiro, and N.R. Rodriguez. "Cooperative multi-thread parallel tabu search with an application to circuit partitioning", Lecture Notes in Computer Science 1457 (2004), 310-331.

2. Aiex, R.M., C.C. Ribeiro, and M. V. Poggi de Aragão. "Parallel cut generation for service cost allocation in transmission systems", Proceedings of the Third Metaheuristics International Conference, (2005), 1-7.

3. Amestoy, P., P. Berger, M. J. Daydé, I. S. Duff, V. Frayssé, L. Giraud, and D. Ruiz (eds), Proceedings of the 5th International Euro-Par Conference (LNCS 1685), (2000), 89-162.

4. Andreatta, A., and C.C. Ribeiro. "A graph partitioning heuristic for the parallel pseudo-exhaustive logical test of VLSI combinational circuits", Annals of Operations Research 50 (1994), 1-36.

5. Andrews, G., and F. Schneider. "Concepts and notations for concurrent programming", Computing Surveys 15 (2000), 3-43.

6. Applied Parallel Research. Applied Parallel Research Home Page, 1999. (http://www.apri.com)

7. T. Cormen, C. Leiserson, R. Rivest, and C. Stein. Introduction to Algorithms. MIT Press, 2001.

8. Motorola ColdFire. Available on-line at http://www.motorola.com/ColdFire

9. IMPACT compiler publications. Available on-line at http://www.crhc.uiuc.edu/IMPACT/index.html