2022 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA) | 978-1-6654-9991-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICAICA54878.2022.9844487

Parallel computing and its applications

Yukai Huang
Applied Mathematics, University of California, Davis, Woodland, CA, 95776, United States
hykhuang@ucdavis.edu

***Abstract*—Parallel computing is the process of running an application or a computation on many processors at the same time. It's a type of computer architecture in which enormous issues are broken down into smaller, typically related components that can be processed all at once. Multiple CPUs communicate through shared memory to complete the task, which then combines the findings. It aids in the execution of big calculations by dividing the enormous problem across several processors. By boosting the available computation capability of systems, parallel computing aids in speedier application processing and job resolution. Most supercomputers work on the basis of parallel computing concepts. Parallel processing is widely utilized in operational settings that require a lot of processing capacity or calculation. This architecture is often housed in a server rack with many processors; the application server divides the computational requests into little pieces, which are subsequently executed simultaneously on each server. The first computer software was built for serial calculation, which can only execute a single instruction at a time. In contrast, parallel computing uses several processors to run an application or computation at the same time. In this paper, fundament theory of parallel computing is discussed.**

Keywords—parallel computing, supercomputers, units

I. Introduction

For computers, when certain actions are needed, the software will experience a series of computation, so-called serial computation, and during this process, the problem will be divided into different series of instructions, where each one of the assigned instructions will be implemented one after another. This entire computation can only be achieved in a single computer, or a single processor, and meanwhile there could be only one instruction being implemented during the computation [1-10].

Now, we are coming to the idea of parallel computing. The definition of parallel computing could be explained in simplicity, that it is a process to tackle a computational issue using more than one compute source at the same time. Unlike serial computing, parallel computing makes it possible to solve the problem in different series at the same time, no need to wait for one-after-one instructions, which greatly saves time. After the problem has been divided into different stages, parallel computing requires that the problem be spitted up into further stages, where multiple processors implement at the same time. For parallel computing, a head cooperator is required for the overall control of such a giant system. Regarding the computation problem, it should satisfy the three properties in the following to be considered appropriate. First, the problem can be divided into different categories that can be tackled concurrently, which is the most important, since otherwise the model of parallel computing fails. Next, under multiple instructions, the problem can be processed at any given time during the computation. Finally, the time cost of resolving this problem using multiple resources should be less than that of a single computer. These are the three main properties for the problem to be held. Aside from the problem itself, the resources we are looking for, multiple computers, can be made up of a single computer with different processors, or a variety of computers linked together by a head network.

The core of parallel computing is to employ parallel computers. So, what are they? Nearly all computers we are using today can be defined parallel from the hardwire perspectives. It is because they share these common features: multiple function units, multiple execution units and multiple hardware threads. Then, by connecting single computers(nodes) together using the network, the cluster of parallel computers forms. Take an example of a common LLNL computer cluster. In this cluster, every compute node is itself a multiprocessor computer, and they are linked together by the infiniband network. Up to now, most giant parallel computers in the world are made up of prestigious vendors who each provide a handful of hardware.

Now, we are coming to the question: Why do we use parallel computers? Our world is incredibly sophisticated and inconceivably massive, making it possible for many things to be interconnected and influenced by each other. These events do not occur one after the other, but rather affect each other to varying degrees. Compared to serial computing, clearly, parallel computing is more appropriate in solving these dilemmas, like climate fluctuations, planetary movements, galaxy formation and so forth. Besides, parallel computing can save a great deal of time since the effort made by a corporation of computer resources could greatly shorten the completion of various work. Moreover, the cost of producing parallel computing is inexpensive since they come from cheap commodities. Parallel computing is also an effective strategy for solving even more complex issues that cannot be simply tackled by the serial method. It provides large memory storage for the problem, which allows for things to operate concurrently on different computers in the system. Furthermore, unlike serial computations, parallel computing could employ the resources everywhere on the network, which provide efficiency even when the local resources are scarce. Hence, with a bunch of these merits, parallel computing has been widely applied in our computers as the building blocks. For example, the laptops we use today are parallel in their multiple processors. Were the serial computation kept running on modern computers, it would be a huge waste to the power of computers.

Given the advantages above, parallel computing has gradually become the future of computing for the pursuing of faster pace, multiple processors, as well as decentralized systems. Up to now, we have experienced an increase of over 50000x in computer performance, which implies that we are entering a new period called the Exascale era.

II. Main body

Who is employing parallel computing? Given its powerful capabilities, parallel computing has, for a long time, been applied in the field of engineering and science, to solve complex mysteries. For instance, it is used in physics, chemistry, atmosphere, geology and so on. Nowadays, the focus of computer science has moved to pursue faster speed in computer performance to solve more intricate issues, which requires processing vast amounts of data in a more sophisticated manner. For instance, it is used in artificial intelligence, financial and economical modeling, data management, virtual reality and so forth. Hence, with all this background information, it is clear to claim that parallel computing has now been widely applied in many different fields and is considered the future way of computer development.

Now, let's take a deep look at the history of parallel computing. The first man who authorized the general requirement for an electronic computer was mathematician von Neumann. According to Neumann, the new electric computer differs from earlier computers in that all the data and process instructions are saved in the memory. Then, the four basic components for a computer were memory, control unit, arithmetic logic unit, and input/output. Memory is where data and instructions are stored. The control unit retrieves the commands/data from the memory, and decodes the instructions, then synchronises the operations in accordance with the sequences to fulfil the task. Arithmetic units perform simple arithmetic operations, and input/output is what we communicate with computers.

Parallel computers can also be categorized into different types, of which Flynn's classical taxonomy, used since 1966, is one of the more widely used classes. According to Flynn's taxonomy, multiple processors are categorized based on their instruction steam and the data stream. For example, the taxonomy includes SISD(single instruction stream and single data stream), SIMD(single instruction stream and multiple data stream), MISD and MIMD, of which each of them can only have two properties: either being single or being multiple. Let's take a closer look at each of these four different types in the classification. Firstly, SISD, which implies single instruction single data stream, consists of only one serial computer, and for it to operate each time, there must be only one instruction at any moment acted by CPU, and the input only includes one stream of data. This type of classification symbolizes how the oldest computers work, and the most common feature of this type of computer is their deterministic execution. Some examples of SISD are early minicomputers, single processor, etc. Secondly, SIMD, single instruction with multiple data streams. This type of computer can deal with different kinds of data on processors, which made them the ideal computers to cope with problems in structure, like images, graphs, etc. The common feature of this type of computer is synchronous and deterministic execution. Some examples are processor arrays like thinking machines CM-2, MP2, and vector pipelines such as IBM9000, C90, ETA10, etc. Nowadays, most of our computers focusing on graphic processing employ SIMD execution. Next, MISD, multiple instructions with a single data stream. This type of computer has not been invented yet, and an imaginable example of such a computer is multiple cryptography algorithms to decode a single message. Finally, MIMD, multiple instructions and multiple data streams. This type of computer is much more competent, and it can be synchronous or non-synchronous, deterministic, or non-deterministic. Our parallel computers right now are examples of using this model. Moreover, most MIMD computers include SIMD as its sub-components.

Now, after introducing Flynn's taxonomy, there are several computing terminologies for parallel computing. CPU, an independent execution unit with its own instruction flow. Node, a self-contained "computer in a box", consisting of multiple CPUs/processors/cores, memory, network interfaces, etc. The nodes are connected via a network to form a supercomputer. SYMMETRIC MULTI-PROCESSOR (SMP), a shared memory hardware architecture where several processors share a single address space and have equal access to all resources - memory, disk, etc. COMPUTATIONAL GRANULARITY, a quantitative measure of the computation to communication in parallel computing: Coarse implies a great amount of computation, and fine means the other side.

Overall, in general, parallel operations tend to be more sophisticated compared to their serial counterparts. Not only do you have multiple instruction streams running at the same time, but also have data moving between them. Also, the cost of complexity can be seen in nearly every aspect in the software development cycle: designing, coding, debugging, maintenance, etc. In terms of portability, parallel computing seems to be better in recent years, but the essential problems within serial computing have rooted in parallel computing as well. The implementation of several APIs can vary in some details even though they are standardized, and sometimes even the code needs to be modified to achieve portability. For resource requirements, the goal of parallel computing is to shorten the execution time in each cycle, but to do this, a greater amount of CPU time is needed. Parallel code may require a larger amount of memory than serial code due to the need to copy data and the overhead associated with parallel support libraries and subsystems. For short-running parallel programs, performance may degrade compared to a similar serial implementation. The overhead costs associated with setting up the parallel environment, task creation, communication and task termination may account for a significant portion of the total execution time for short runs. Another key feature of parallel computing is its scalability, which is either strong or weak. Strong scalability, advocated by Amdahl, indicates that with more processors included, the size of the issue remains unchanged, and the perfect time of solving a problem is 1/P. Weak scalability, offered by Gustafuson, stated that the size of the problem per processor remains constant with an increasing number of processors. The overall problem size is in proportion to the number of processors employed.

Now, let's talk about the parallel computer memory architecture. Parallel computers of shared memory differ greatly, yet typically hold in common the fact that all processors can access all memory as a global address space. Plural processors can operate separately but share the same memory resources. Variations in the location of memory influenced by one processor are visible to all other processors. History shows that shared memory machines have been categorized into UMA and NUMA based on memory access times. In terms of UMA, Uniform memory access, it is mostly shown by SMP machines recently, and has the same processors. It also has equal access to memory. Sometimes referred to as CC-UMA - Cache Coherent UMA, which means that if a processor renews a place in shared memory, every processor will know about the update. Cache coherency is achieved at the hardware level. In contrast, NUMA, Non-Uniform memory access is often referred to as machines made by connecting two or more SMPs. For NUMA, only SMP could gain access to another SMP, and some processors have varying access time to memory. Memory access is slower across the link, and it can also be called CC-NUMA - Cache Consistent NUMA if it keeps cache coherence. Overall, the advantages of such architectures provide a friendly environment for users on the global address scale, and it made data sharing identical and fast-paced at the same time by the proximity between CPU and memory. On the other hand, the main drawback lies in the absence of scalability between memory and CPU. Appending more CPUs can increase the amount of flow on the shared memory-CPU path geometrically, and for cache coherent systems, the flow associated with cache/memory management geometrically. Programmers are accountable for the synchronous structures that guarantee " proper" global memory access.

Another memory architecture is the distributed memory. A common feature between shared memory and distributed memory is they both have an internal network that builds for communication to interior processor memory. For distributed systems, each processor has its own local memory, so addresses in one processor do not correspond to another, making the term global address nonsense among all processors. Therefore, the modifications of one processor will not influence the others since they are all independent processors, and hence cache coherence does not apply here. If there is a need to use the data in another processor, programmers need to decide what communication is needed. The advantage of distributed memory is that with an increasing number of processors comes a greater amount of memory, and each processor can access its memory without interfering with the other. In addition, distributed memory is cost-effective as it employs commodity processors. However, programmes are accountable for the majority of data communication among processors and mapping available global memory-based data structures to this kind of memory organization can be difficult. Moreover, it takes more access time for the node in remote areas compared to the node local data.

Next, hybrid distributed-shared memory. This type of memory structure is widely employed on our fastest and most advanced computers nowadays, in which distributed memory and shared memory consist of the two major components in this architecture. Shared memory can be a GPU or a shared memory machine, while distributed memory builds connections between different GPU and shared machines. Hence, under this structure, transmitting data from one to another requires net connection. Up to now, this memory architecture is considered the future way of computing. Its pros and cons are the same as those of distributed memory and shared memory.

Given all these memory structures, let's now look at some parallel programming models. Some prevailing models are Shared Memory without threads, Distributed Memory/Message Passing, Hybrid, SPMD and MPMD. Overall, these models lie above the hardware and the memory structure, which indicates that every model can be applied to the underlying hardware. For example, a shared memory model on a distributed memory machine. It also has another name called Kendall Square Research (KSR) ALLCACHE approach. In theory, machine memory is dispersed among different connected processors, but the users could only perceive a shared global address. Hence, this method is also termed as "virtual shared memory". Another approach is a distributed memory model on a shared memory machine. In this method, it employs a previous model called MPI on SGI origin 2000. SGI Origin 2000 utilises a shared memory structure of the CC-NUMA type, where individual tasks can access a global address space distributed across all machines directly. Nevertheless, to send and retrieve information using MPI, as is commonly done on networks of distributed memory machines, has been implemented and is commonly used. There are no best models of these two and determining which one is better depends on personal preference and whether they are available.

Given the two models above, there are some of their applications in the real world. The first one is a shared memory model (without threads). In this system, each program and task are located at the same space and writing and reading data is asynchronous. There are a variety of locks, semaphores, in which they serve to manage the access to memory, avoid arguments, and protect the system from experiencing deadlocks. By far, this model is probably the simplest one, and a merit of this model is that programmers do not need to clarify the connections between data and programs. However, interpreting data and administrating the data locality become very difficult based on its functionality. It is assumed that maintaining data in the local area of the process saves memory accesses, cache flushes and bus traffic, but controlling data locality for ordinary people like us can be very hard to understand. To implement this model on a single shared memory machine, operating systems and hardware are required for computational programming.

Another model is called Threads model, which lies in the classification of shared memory programming. In this model, a single process could have many different execution approaches. For instance, given a main program m.output, in which m.output did some serial work and then generated some threads which then can be executed concurrently. For each thread, they all benefit from the global m.output in that they share the memory storage of the entire program. The implementation of such a model often includes several execution approaches that are mandated by the source code among parallel machines, and a collection of directives for the editor built into the sequential or paralleled source code. In both scenarios, parallelism is determined by the programmers.

Then, Distributed memory model/Message Passing Model. The feature of this model is that computation work comprises several tasks that employ their own memory, and all those tasks could rely on the same machine or build upon a number of different machines. The communication of data in this model depends on sending and receiving messages, and meanwhile each data transaction requires the operation of sending messages and the one receiving messages.

To assess the cost of an algorithm, general work is needed. Therefore, we hypothesize that the cost of a computer is related to the number of processors of that computer, and the expense of buying of a computer is also proportionally related to the cost of a computer. Therefore, the overall computation of a computer is equal to the number of processors multiplied by the time needed for comprehensive computations. In the end, this product is the work of a particular algorithm. Here, we define the term Fundamental lower bound Tp. For base case, the total work is distributed evenly among p processors, and thereby the time required by each processor is also evenly divided. Then, the lower bound of Tp is given by T1/p <= Tp, where the best phenomenon is shown by this equation. Next, let's move on to theorize the upper bound of Tp.

So, the question is what happens when there are infinitely many processors? The general assumption is that the time for computation is close to zero; however, this is not often the case. Now, we define that T1, Tp, and T infinity by the equation T1/p <= Tp <=T1/p + T infinity. It is known that T1/p achieves its maximum, so T infinity itself tells us how parallel an algorithm is. This theorem is fabulous in that if we realize how our algorithm performs on an infinitely many processors, we will get to know how it performs on a given number of processors. To prove this, on the level I of DAG, there are mi operations. T1, the total work of computation, is equal to the sum of mi. For each I of DAG, the time of each processor is defined as Tp = mi/p <= mi/p +1. From this equality, we know that when all lower operations are completed, the processors have no interdependencies. Therefore, we can then assign the same operations to processors. If the number of processors is not divisible by p, then a wall-clock cycle is needed so that there are some processors being employed in parallel. Hence, in work-depth model, our purpose is to make algorithms that can be applied to an infinite number of processors, which minimizes T infinity, and thereby pushing us closer to our optimal bound of T1/p.

III. Conclusion

In this paper, we've discussed what parallel computing is in our world of computing, its history, its components, and a variety of computing models generated from parallel computing. It is noticeable that parallel computing will become our future way of computing, not only because it is much more appropriate for analyzing unsolved mysteries like climate changes, nebular movement, and so forth, but also because compared to serial computing, it provides us with new insights and efficiency in transmitting data and utilizing the power of computers. Soon, there will be more hybrid models included during computation for the sake of costs and efficient data transportation. Overall, parallel computing is an integral part of computing development in this era.

References

[1] Adve, S. et al. Parallel Computing Research at Illinois: The UPCRC Agenda. White Paper. University of Illinois, Urbana-Champaign, IL, Nov. 2008

[2] Agarwal, V., Liu, L.-K., and Bader, D. Financial modeling on the Cell broadband engine. In Proceedings of 22nd IEEE International Parallel and Distributed Processing Symposium (Miami, FL, Apr. 14--18, 2008).

[3] Alexander, C. et al. A Pattern Language: Towns, Buildings, Construction. Oxford University Press, 1997

[4] Asanovic, K. et al. The Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View. UCB/EECS-2008-23, University of California, Berkeley, Mar. 21, 2008.

[5] Asanovic, K. et al. The Landscape of Parallel Computing Research: A View from Berkeley. UCB/EECS-2006-183, University of California, Berkeley, Dec. 18, 2006.

[6] Bader, D.A. and Patel, S. High-performance MPEG-2 software decoder on the Cell broadband engine. In Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (Miami, FL, Apr. 14--18, 2008).

[7] Buschmann, F. et al. Pattern-Oriented Software Architecture: A System of Patterns. John Wiley&amp;Sons, Inc., New York, 1996.

[8] Datta, K. et al. Stencil computation optimization and autotuning on state-of-the-art multicore architectures. In Proceedings of the ACM/IEEE Supercomputing (SC) 2008 Conference (Austin, TX, Nov. 15--21). IEEE Press, Piscataway, NJ, 2008.

[9] Demmel, J., Dongarra, J., Eijkhout, V., Fuentes, E., Petitet, A., Vuduc, R., Whaley, R., and Yelick, K. Self-adapting linear algebra algorithms and software. Proceedings of the IEEE, Special Issue on Program Generation, Optimization, and Adaptation 93, 2 (Feb. 2005), 293--312.

[10] Engler, D.R. Exokernel: An operating system architecture for application-level resource management. In Proceedings of the 15th Symposium on Operating Systems Principles (Cooper Mountain, CO, Dec. 3--6, 1995), 251--266.