Assess Current Research in Big-Level Parallelism

Yiqiao Yin

Computer Science, PhD

TIM8121 Distributed Algorithms and Parallel Computing

Professor William Souza

September 2022

Assess Current Research in Bit-Level Parallelism

This assignment investigates the bit-level parallelism. As one of the earliest forms of parallelism used in computer science, bit-level parallelism has been a fundamental concept for major courses of computer science departments. This assignment conducts research and investigates the current trends in bit-level parallelism. In addition, the paper discusses a variety of different resources of how the target concept benefits the entire computer science community. The paper makes presentations of diagrams and graphs to demonstrate the validity of the idea.

Literature Review or Background

The story started in the early 1960s at IBM where it had four incompatible computer lines. The computer lines had their own design of the architecture. It is called instruction-set architecture (ISA). The exploration started when engineers dreamed about having a computer chip to execute jobs on all lines of the computer together correctly. An important concept to introduce is called Read Only Memory (ROM), which is a type of memory commonly used in computers and computer engineering (Wilkes, 1958). The ROM stores data and acts as non-volatile memory system. Suggestions have been made to propose a novel design to reassemble programming components at a relatively low but detailed level. This is called microinstruction (Patterson, 2018). This led to the major development and the success of the IBM era. It projected the sales price for the lowest model that uses 4000 microinstructions 50 bits to be above $133,000 in 1964 (Patterson, 2018). At today's price, this range is well above $1 million dollars (Patterson, 2018).

Distributed Computing System

The motivational question in distributed computing system in computer engineering is to answer the question of the number of jobs computation can be executed using a domain at the

same time. There is a crucial assumption in this scenario. If the jobs are sequentially designed, the concept of "at the same time" cannot be held. Hence, for this crucial assumption states that the jobs cannot be sequentially designed. For example, a for-loop can be considered a design of the type of task such that this assumption holds. The reason is because each step inside of a for-loop is generally considered to be independent. In other words, any two steps in the for-loop have no affect to each other. This is the type of scenario that parallelism is allowed. However, in algorithmic design, sometimes the downstream job cannot happen if there is a job upstream that is not finished. For example, a hierarchy of if-statement is usually considered not appropriate for parallelism. To ensure the clarity of this assumption, the following examples are presented. When the programming environment allows parallelism, the capacity of the system is investigated to establish the job function. This is an area where the power of choices arises (Han, 2020). Henceforth, the assignment of such task for choosing the power of choices is an important area for designing the capacity of parallelism.

Scenario: An experiment that can be used for parallelization

First, let us consider a Monte Carlo simulation to reproduce the numerical estimate of "Pi". The simulation of this experiment is presented in Figure 1. In Figure 1, two plots are presented. The left plot draws sample data from a uniform distribution, i.e., $Uniform(0,1)$. The parameters of the uniform distribution ensures that the data points drawn fall inside of a unit square with size 1. The algorithm can check the distance between the coordinate of each data sample and the origin, i.e., $(0, 0)$. If the distance is less than 1, then the dot is marked in red. Otherwise, the dot is marked in black. Every sampled experiment, an estimate can be computed by counting the number of red dots over the total number of dots. Then an equation can be set up by using $r = \frac{1}{4}\pi$, where r is the ratio of red dots over total number of dots. This allows us to

reverse engineer an estimate of the π value, i.e., $\pi = 4r$. This estimate can be plotted on the graph on the right of Figure 1. The simulation of the samples is independent experiment. Hence, this is process is allowed to be parallelized.

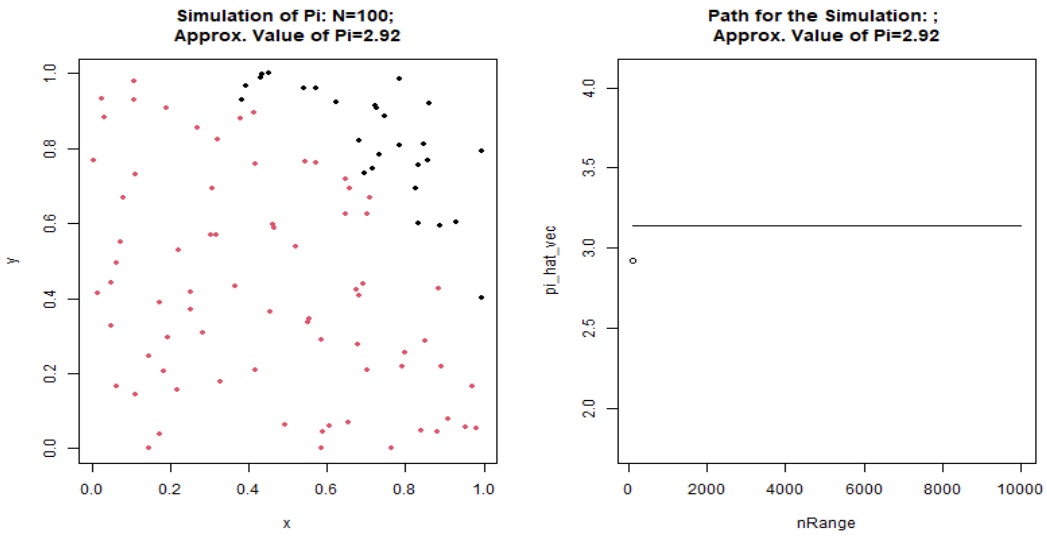

Figure 1. Monte Carlo Simulation

Scenario: An experiment that can NOT be used for parallelization

Second, let us consider an experiment that cannot be used for parallelization. The Fibonacci series is a famous example that uses the sum of the previous two numbers to define the current number. Mathematically, the series can be stated $X_t = X_{t-1} - X_{t-2}$. In other words, the series is sequentially defined. If a random integer is picked and the goal is to compute the Fibonacci sequence, the number cannot be computed unless all the previous numbers are known. Hence, the concept of "at the same time" breaks and will not hold in this scenario. Multiple-node processors can be used, but a node cannot start its computation unless all premises are known. In Figure 2, a sample code snippet is written and the while-loop inside the definition cannot be executed on multiple different nodes. This is a scenario where parallelism cannot be used.

```
def this_fibo(nterms):
    cnt = 0
    n = 0
```

```
    first_num, sec_num = 0, 1
    while cnt < nterms:
        n = first_num + sec_num
        first_num = sec_num
        sec_num = n
        cnt += 1
    return n
```

Figure 2. Code for Fibonacci Sequence (in Python)

Computing Performance

There are many computational problems require advanced programming knowledge and sophisticated design of parallelism. This is an important step to achieve the desired results for computer scientists and nowadays for data scientists. The parallelism is not only about achieving high speed but also decreasing the size of the problems to increase energy efficiency. This is especially beneficial because parallel computing can reduce the time-to-solution and increase the energy efficiency in a computer programming (Robey, 2021). In addition, the surprising results today have shown that the parallel computing is the most important domain to improve upon in some of the largest computing systems. It is even made available for consumer laptops and even mobile services at present day. This creates an ecosystem that the computers are vastly accessible almost nearly everywhere for developers and consumers for a variety of different purposes.

The most beneficial for data scientists is the reduced run-time of computer programs using multiple cores. The reduction of a computer program not only implies increased speed and training time, but also imply potential training capacity parallelism can deliver. This means data scientists could potentially achieve much better outcome and deliver faster and more accurate data-driven business insights.

The energy consumption is related to the cost of the programming, and it is directly related to what prevents business from preserving the revenues created from business activities. The energy consumption takes the following form of equation

$$P = (N\ Processors) \times \left(R \frac{Watts}{Processor}\right) \times (T\ hour)$$

where the notation P is the energy consumption, N is the number of processors, R is the thermal level capturing the design power, and T is the time consumption used to run a desired software. Commonly, Graphical Processing Units (GPU) have higher energy consumption than Central Processing Units (CPU) (Robey, 2021).

Current Trends

The bit-level parallelism or bit-parallelism is the lowest level of parallelism (Petri, 2012). The concept of "bit" comes from the logical expression where "1" and "0" are the only available outcome in the computer programming. This could be used as logical component but can also be used to express words and meanings. There are operational units that incorporate single inputs, and these are called single input processing units or SPU. The number of SPUs is dependent on the number of input bits for parallelism (Petri, 2012). The processing units are also dependent on the architecture of the design. Usually, the SPUs are designed in parallel. There are multiple inputs and multiple outputs.

One advantage of bit-level parallelism and potential popular research is to create designs with joint neighbors where each neighbor is a node for operation such that independent SPUs can be set in place to create double computation. The attractive perspective of bit-level parallelism is its wide range of applications in mobile services (Petri, 2012).

In addition, bit-level parallelism can be extended into block-level parallelism (Petri, 2012). A cluster of bits can be set together, and this can be considered as a unit block (Petri,

2012). This block can act as a single unit with the capacity of 4x or 8x that of a single bit (Petri, 2012). The block can be used to execute jobs together independently to allow the decoder process substantially more convenient when executing jobs with higher dimensional arrays in the data.

Block-level parallelism can also be used to stream data online or remotely (Petri, 2012) (Patterson, 2018). This was considered one of the most amazing discoveries in modern day computer science and the idea is based on dividing data stream into multiple currents. Based on the block-based unit, the data stream can be accessed using the blocks instead of bit-level. This allows the parallelism to process images with higher speed (Petri, 2012). For analyzing this component and measure the performance, denote a block to have N_{block} bits and a block-based unit (BBU) can start running jobs with the foundation of bit-level parallelism. The resulting output of is

$$R_{total} = N_{BPU} \times R_{BPU},$$

where the R is the throughput and the equation computes the total throughput from BPU blocks and the throughput of a single BPU. The jargon "throughput" refers to the amount of information or material passing through a system. In this case, it can refer to data or energy. The total throughput can be written as

$$T_{BPU} = T_{BP} + N_{block} \times \frac{N_{BPU}}{R_{total}},$$

and the latency T_{BPU} indicates the length of the time consumed for the passage to transfer its data from one end to the other.

Future Research

This section discusses the technology that builds up the concept of bit-level parallelism. This article has discussed bit-level parallelism (BLP) that performs using multiple different bits.

The next level up is to use instruction-level parallelism (ILP) which executes logically sequential instructions using pipeline which further improves the speed and reduces energy consumption (Marcuello, 1998). The higher tier is thread-level parallelism (TLP) which utilizes multiple different BLPs or even ILPs (Marcuello, 1998) (Kumar, 2006). This type of machine conducts operations discretely and integrate many different multi-core processors at the same time. The last level of parallelism is data-level or also known as data-level parallelism (DLP) (Marcuello, 1998). This is no longer lingering on the node or the design of multi-core processor but on the design of algorithms and course of operations (Kumar, 2006). The data formed by arrays of numbers need to be processed into appropriate channels at the right time using the correct order. It not only requires the computer scientists to be familiar with coding but also need to be familiar with the data itself.

Conclusion

The article investigated the current trends in bit-level parallelism by starting with the premises required for the parallelism operation. Two scenarios are proposed, and the difference of job architecture is discussed for parallelism to hold. The article then discussed the trend and future research in regard to bit-level parallelism and provide overview of each direction.

References

Han, P. W. (2020). Capacity analysis of distributed computing systems with multiple resource types. *2020 IEEE Wireless Communications and Networking Conference (WCNC)*, 1-6.

Kumar, R. T. (2006). Core architecture optimization for heterogeneous chip multiprocessors. *2006 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 23-32.

Marcuello, P. G. (1998). Speculative multithreaded processors. *Proceedings of the 12th international conference on Supercomputing*, 77-84.

Patterson, D. (2018). 50 Years of computer architecture: From the mainframe CPU to the domain-specific tpu and the open RISC-V instruction set. *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, 27-31.

Petri, M. (2012). Latency impacts of different parallelism levels in data-flow architectures. *The 15th International Symposium on Wireless Personal Multimedia Communications*, 500-504.

Pugsley, S. H. (2014). NDC: Analyzing the impact of 3D-stacked memory+ logic devices on MapReduce workloads. *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 190-200.

Robey, R. &. (2021). *Robey, R., & Zamora, Y. (2021). Parallel and high performance computing.* Simon and Schuster.

Wilkes, M. V. (1958). The design of the control unit of an electronic digital computer. *Proceedings of the IEE-Part B: Radio and Electronic Engineering*, 121-128.

Appraise Multithreading Techniques

Yiqiao Yin

Computer Science, PhD

TIM8121 Distributed Algorithms and Parallel Computing

Professor William Souza

September 2022

Appraise Multithreading Techniques

There are many different techniques available for instruction-level parallel computation or also known as ILP. Common techniques include but not limited to pipelining, superscalar execution, out-of-order execution, renaming registration, and speculative execution. Recently, scientists have been focusing on the development of multithreading techniques for ILP. This paper investigates and conducts research to review the supporting evidence in the literature of multithreading techniques and how this can be applied to ILP. In addition, the paper addresses the advantages and disadvantages for this direction. Literature review is provided to introduce this field and its related existing methods.

Literature Review or Background

There is growing interest in the development of multiple-issue machines and the requirement for heavily pipelined machines are at the center of interest for computer science community (Jouppi, 1989). Out of many existing architectures, the superscalar machine is a popular one that can execute multiple independent instructions in one computation cycle (Agarwala, 1987). As its name suggests, the design handles data structures as arrays with the n^{th}order. The superscalar executes instructions follow an array design while each array has its own base cycles. In order to truly achieve real parallel behavior, the design utilization the n^{th}degree of order and sets a unit of degree(s) for a base at a time. In addition, a superpipelined machine can also get the job done, but it executes just one instruction in a computation cycle while the cycle periods are shorter (Jouppi, 1989). The superpipelined machines take advantages of ILP in a unique way (Jouppi, 1989). Suppose a design has an order of n. It seems like that the parallelization can be achieved using Limitations have been raised that the basic parallel computing using block designs seldom produce performance that goes beyond three or four blocks on average (Jouppi, 1989) (Smith, 1989) (Tjaden, 1970). This is a serious accusation that

deserve our attention, because it brought out the non-linear nature of increasing the design with the belief that similar levels of performance boost can have. To further address the issue, it is important to first analyze the effects of the compiler optimization (Jouppi, 1989). There are several available strategies useful for ILP on compilers (Jouppi, 1989). First, the loops of array-like design can be paralleled (Allen, 1987). For the less parallelizable design, the loops can be unrolled and then scheduled to be traced (Ellis, 1986) or can be pipelined using software (Charlesworth, 1981). However, due to memory latency and processing, the barrier of multi-core processor has been facing challenges to produce consistent and robust performance, which leads to the development of multithreading as the major choice of the architectural design of multi-core processors (Saavedra-Barrera, 1990).

Multithread Architectures for Parallel Computing

The architectural challenges posed grave danger to deploy scalable modules in production due to the memory latency initiatives and delayed computation speed. For this purpose, two alternatives are studied: avoid the latency problems or tolerate it (Saavedra-Barrera, 1990).

The strategy to avoid the latency issue or cached memory was developed because the copies of the data structures in cache was produced as the user required it, which led to the replication of the memory issue (Goodman, 1983) (Agarwal, 1988).

Alternative strategies may also exist and be adopted. This philosophy gave birth to multithreading designed architectures. The concept of multithreading, as the name suggests, comes from the nature that each processor can support more than one threads (Saavedra-Barrera, 1990). The data workflow goes through the processor and executes the task given by its human commander. There is a queue in the processor that loads up a list of unfinished tasks waiting to be executed. In the front of the queue, the task gets executed and a signal is passed through the

processor. Though this workflow is straightforward, it is still yet unclear how a processor waits during a reference point and how does a thread execute the job. To address these concerns, the next session of the paper discusses a small quantitative model to measure this design issue.

A quantitative model to understand multithread processor

In a conventional processor with a single thread, the waiting happens during a remote reference. In a multithreaded processor, the wait suspends the current issue, and a switch is flipped on to turn a task from one to the other. The key is to handle "idle" situation in an optimized manner. Hence, the objective is to maximize the time of the fraction when the processor is in the middle of executing a task. Let us denote the efficiency of the processor to be ϵ. The efficiency of the processor is, hence, given by this formula Equation 1. Efficiency of the Processor,

Equation 1. Efficiency of the Processor

$$\epsilon = \frac{Busy}{Busy + Switching + Idle}$$

where the "Busy", "Switching", and "Idle" refers to period measured over certain intervals of state that the processor behaves. The processor is considered "Busy" if it is during the execution of a task. The processor is considered "Switching" if it stops an execution of the job on one thread and then starts the job execution on another thread. The processor is "Idle" if no real execution is happening. Based on the foundational concepts above, a processor with a single thread can execute a job until a signal from its human commander is issued. Let us denote this issue by I. The "Idle' period starts until the job is sent to the chip and starts execution. Let us denote the completion period L. In terms of the notation in Equation 1. Efficiency of the Processor, the R and L represent "Busy" and "Idle", respectively. Hence, the efficiency of the processor with a single thread is given by

Equation 2 Efficiency of the Processor

$$\epsilon_1 = \frac{R}{R+L} = \frac{R/R}{\frac{R}{R}+\frac{L}{R}} = \frac{1}{1+\frac{L}{R}}$$

This formula shows that the efficiency of the processor increases "Idle" period decreases or "Busy" period increases.

Design a machine with multithreaded structure

In the above, this paper has discussed the desire of building a multithreaded architecture. This section of the paper discusses the potential approach of how a multithreaded architecture can be built. First, a memory latency cycle needs to be selected and there can be 32, 64, or 128 (Saavedra-Barrera, 1990). Based on the complexity and efficacy of the project structure, the design of the most aggressive setup attempts to make a switch of cores to execute the jobs immediately. This allows the fastest switching rate, which reduces the "Idle" time (length of L) in the Equation 2 Efficiency of the Processor (Saavedra-Barrera, 1990). The next tier provides a medium to aggressive approach which is designed based on cycles 1, 4, or 6 cycles for the multi-core architecture. This should reduce the miss ratio in the switching cost by 1.1%, 2.2%, and 3.1%, respectively (Saavedra-Barrera, 1990) (Goodman, 1983). Overall, the central idea is that a series of models can be designed and built to produce higher and more consistent performance for multithreaded processors.

Performance Analysis for Various Parameters in Single- vs. Multi-Threaded Modes

This article has discussed variety of different components possible from architectural perspective. However, it still requires some empirical evidence to truly convince our readers that the multi-core design is advantageous.

To investigate and shed some light on the performance of single- vs. multi-threaded design, it is important to deploy a model to a real data using multiple different Central

Processing Unit (CPU) and Graphical Processing Unit (GPU) (Kochura, 2017). Fortunately, the literature has these studies been done already using the famous MNIST dataset (Kochura, 2017). The dataset consists of a training set and a test set. The training set has 60,000 images and the test set has 10,000 images. Each image is a handwritten digit with size 28 by 28. This means an image has $28 \times 28 = 784$ pixels. The machine learning methods used in this experiment is a deep neural network (LeCun, 2015). The reason a deep neural network model can learn the optimal parameters to make predictions well is based on the idea of gradient descent (Amari, 1993) which requires many iterations of the algorithms to achieve optimal outcomes. This is a procedure where parallel computing can be adopted and can largely increase the performance and the efficiency of training. The most significant result comes from an experiment between an Intel Core i5 CPU and Intel Core i7 CPU where the i& core has faster capacity and computing speed. It turned out that to achieve the same loss an i7 core used 65 iterations where an i5 core used 103 iterations (Kochura, 2017). In addition, the i7 core can provide training speed with 80,000 observations per second while the i5 core could only provide about less than 25,000 observations per second, a significant difference when the data is larger (Kochura, 2017).

Conclusion

The article investigated the procedure of developing multithreading techniques. The paper started with literature review of the general trend of developing multi-threaded cores in the field of parallel computing. In addition, the paper provided some analysis of the architecture and the mathematical form of how performance is measured. Then the paper discussed and landed on a review of an application of how machine learning applications are affected using different chips for computation candidate.

References

Agarwala, T. &. (1987). High performance reduced instruction set processors. *IBM Thomas J. Watson Research Center Technical Report# 55845*, 249-260.

Allen, R. &. (1987). Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 491-542.

Charlesworth, A. E. (1981). An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family. *Computer*, 18-27.

Ellis, J. R. (1986). Bulldog: a compiler for VLSI architectures. *Mit Press*.

Jouppi, N. P. (1989). Available instruction-level parallelism for superscalar and superpipelined machines. *ACM SIGARCH Computer Architecture News*, 272-282.

Smith, M. D. (1989). Limits on multiple instruction issue. *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, 290-302.

Tjaden, G. S. (1970). Detection and parallel execution of independent instructions. *IEEE Transactions on computers*, 889-895.

Code a Program for a Multicore Processor

Yiqiao Yin

Computer Science, PhD

TIM8121 Distributed Algorithms and Parallel Computing

Professor William Souza

September 2022

Code a Program for a Multicore Processor

The analysis of processor performance has attracted a lot of interests in the literature. The research heavily focused on the performance of dual core or multi-core processor architectures (Ojeyinka, 2015). This is especially valuable for the development of Intel processor architecture over time where most buyers were buying computers on the spot when walking in the mall or shopping area. The performance system is extremely necessary for a consumer and not many tests are verified and publicly available (Ojeyinka, 2015) (Reinders, 2007) (Russell, 1999). To quantify the analysis of multi-core processor, it is important to note that the performance is largely dependent on a variety of different factors such as design architecture, experimental design, operating systems, different types of compilers, implementation of software program, and so on (Ojeyinka, 2015).

In this homework assignment, the task designs a software algorithm to investigate the BMP file format or the bit map image file. This is a popular type of digital format that stores the image This type of format usually works around a double for-loop or nested for loop. This was originally designed by Microsoft (Kakugawa, 2000). The formal name is called a device-independent bitmap or DIB. It is a format that is used to create the definition of bitmaps in a various of color resolutions (Kakugawa, 2000). The file structure consists of fixed-size definitions with rows and columns of pixels being well defined within the range of image frame. Once defined, the pixels forma arrays and the arrays form a slice of greyscale of scheme taking brightness level from a color category. The information, described by pixel levels, is then saved by byte sized image file and there can be different byte sizes (Bourke, 1998).

Definition of Arguments

The definition of arguments can be tricky. One needs to note that in C++ the arguments are defined using specified data types. The argument of a C++ program is regarding to the values that are passed into a function when the function is called upon. The data type of the arguments is defined upstream in program unless downstream the data type is redefined. Usually, the common practice is to define the data types in the beginning of the code and the data types remain the same throughout the code to be consistent. First, we need to define a "write_image_file" function. This is the function that is doing the heavy lifting. Major parameters will have to be defined. These parameters are file width, file height, image size, bit per pixel, image size in byes, number of colors, and so on.

Each image is assumed to have a fixed width and height. These concepts are denoted as $width$ and $height$. These two arguments define the size and the byte-size of the image file. The number of pixels for a two-dimensional array that stores the greyscale or brightness of a color scheme forms a single slice of image with size $width \times height$. The basic colors of computer science are red, green, and blue. Hence, a colored image would have to have three two-dimensional arrays. The total number of pixels would be $width \times height \times 3$. Each pixel would take values of greyscale or also known as the brightness level from 0 to 255 where 255 is the brightest and 0 is the darkest. The brightness level or the greyscale can be used on a color scheme. At a particular index of row and column, there are three pixel-values where each pixel value refers to a particular color scheme, i.e., Red, Green, and Blue, respectively.

Algorithmic Design

The algorithm design for converting the image has many different versions. The naïve version is to check every pixel and invert the number of the values from black to white and vice versa all on one core. This is the slowest and the algorithm can simply be a nested for-loop.

There are three for-loops. The first for-loop investigates every element of the $width$. The second for-loop investigates every element of the $height$. The third for-loop investigates every color. Hence, the space to search through each pixel is precisely $width \times height \times 3$.

Alternatively, the algorithm design can utilize 2 or multiple cores. For a duo-core processor, we can allocate a core to each slice of the image. Whichever core finishes the slice first can start working on the third slice of the image. There are only three slices in total, because there are only three color-schemes. The workflow is demonstrated in Figure 1. Multi-core Processor Design.

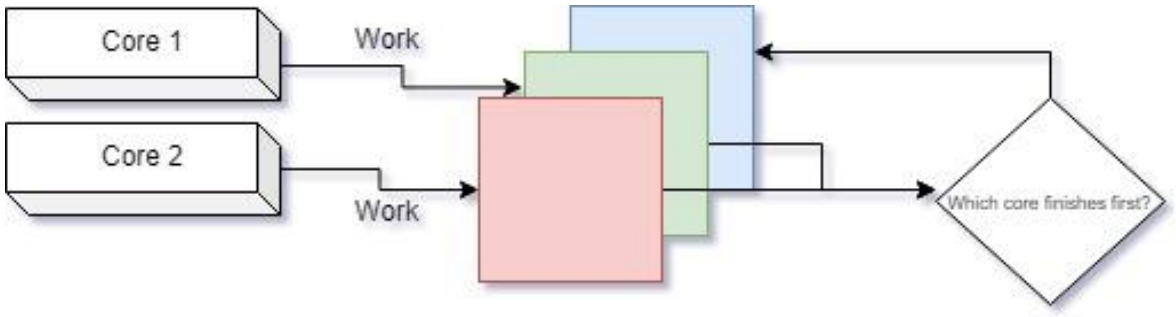

Figure 1. Multi-core Processor Design

Conclusion

This task for this week focuses on the design of a multicore processor and the assignment code a program for a multicore processor on inverting a colored image. The paper proposes a design of the algorithm and code snippet is presented in a separate C++ file.

Bibliography

Bourke, P. (1998). Bmp image format. *BMP Files*.

Kakugawa, H. (2000). A Device-Independent DVI Interpreter Library for Various Output Devices. *Proceedings of the TUG 2000 Conference*, 102-107.

Ojeyinka, T. O. (2015). Performance Analysis of Dual Core, Core 2 Duo and Core i3 Intel Processor. *International Journal of Computer Applications*.

Reinders, J. (2007). Intel threading building blocks: outfitting C++ for multi-core processor parallelism. *O'Reilly Media, Inc.*

Russell, M. K. (1999). Testing on computers: A follow-up study comparing performance on computer and on paper. *Boston College*.

Supplement

Please see the code snippet below.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>
#include "bmp.h"

const int image_width = 512;
const int image_height = 512;
const int image_size = 512*512;
const int pixel_number = 255;

void write_image_file(unsigned char* d, int width, int height){
    struct bmp_id id;
    id.magic1 = 0x42;
    id.magic2 = 0x4D;

    struct bmp_header header;
    header.file_image_size = width*height+54 + 2;
    header.pixel_offset = 1078;

    struct bmp_dib_header head;
    head.header_image_size = 40;
    head.width = width;
    head.height = height;
    head.num_planes = 1;
    head.bit_pr_pixel = 8;
    head.compress_type = 0;
    head.data_image_size = width*height;
    head.hres = 0;
    head.vres = 0;
    head.num_colors = 256;
    head.num_imp_colors = 0;

    char padding[2];

    unsigned char* arr = (unsigned char*)malloc(1024);
    for(int c= 0; c < 256; c++){
        arr[c*4] = (unsigned char) c;
        arr[c*4+1] = (unsigned char) c;
        arr[c*4+2] = (unsigned char) c;
        arr[c*4+3] = 0;
    }


    FILE* fp = fopen("out.bmp", "w+");
    fwrite((void*)&id, 1, 2, fp);
    fwrite((void*)&header, 1, 12, fp);
    fwrite((void*)&head, 1, 40, fp);
    fwrite((void*)arr, 1, 1024, fp);
    fwrite((void*)d, 1, width*height, fp);
    fwrite((void*)&padding,1,2,fp);
```

```
    fclose(fp);
}

unsigned char* fileread(char* filename){

    FILE* fp = fopen(filename, "rb");

    int width, height, offset;

    fseek(fp, 18, SEEK_SET);
    fread(&width, 4, 1, fp);
    fseek(fp, 22, SEEK_SET);
    fread(&height, 4, 1, fp);
    fseek(fp, 10, SEEK_SET);
    fread(&offset, 4, 1, fp);

    unsigned char* d = (unsigned char*)malloc(image_sizeof(unsigned char)*height*width);

    fseek(fp, offset, SEEK_SET);
    //We just ignore the padding :)
    fread(d, image_sizeof(unsigned char), height*width, fp);

    fclose(fp);

    return d;
}

void flip_image_horizontal( unsigned char *d, unsigned int columns, unsigned int rows ) {
    unsigned int left = 0;
    unsigned int right = columns;

    for(int r = 0; r < rows; r++){
        while(left != right && right > left){
            int temp = d[r * columns + left];
            d[(r * columns) + left] = d[(r * columns) + columns - right];
            d[(r * columns) + columns - right] = temp;
            right--;
            left++;
        }
    }
}


int main(int argc, char** argv){

    if(argc != 3){
        printf("Useage: %s image thread_num\n", argv[0]);
        exit(-1);
    }
    int thread_num = atoi(argv[2]);

    unsigned char* image = fileread(argv[1]);
    unsigned char* new_img = malloc(image_sizeof(unsigned char) * image_size);


    //flip_image_horizontal(image,image_width, image_height);
/*  for(int i=0;i<image_size;i++)
    {
        printf("%d\n ",image[i]);
    } */
```

```
    int* invert = (int*)calloc(image_sizeof(int), pixel_number);
    int image_val;
#pragma omp parallel for num_threads(thread_num), private(image_val)
    for(int i = 0; i < image_size; i++){
        image_val = image[i];
#pragma omp critical
        invert[image_val]++;
    }


    float* new_temp = (float*)calloc(image_sizeof(float), pixel_number);
    int i,j;
#pragma omp parallel for num_threads(thread_num) schedule(static,1),private(j)
    for(i = 0; i < pixel_number; i++){
        for(j = 0; j < i+1; j++){
            #pragma omp atomic
            new_temp[i] += pixel_number*((float)invert[j])/(image_size);
        }
    }


#pragma omp parallel for num_threads(thread_num)
    for(int i = 0; i < image_size; i++){
        new_img[i] = new_temp[image[i]];
    }

    write_image_file(new_img, image_width, image_height);
}
```

Distributed Computing Survey Paper

Yiqiao Yin

Computer Science, PhD

TIM8121 Distributed Algorithms and Parallel Computing

Professor William Souza

September 2022

Distributed Computing

The literature has examined a wide range of topics in distributed computing (Waldo, 1996) (Birman, 1993) (Kshemkalyani, 2011) (Prasad, 2015) (Gargees, 2020). Distributed computing is a modular system such that the components of the software blocks are shared amongst many different computers (Kshemkalyani, 2011). In other words, it is a network of systems that connect many different nodes together under one unified hierarchy. There is a certain geographic nature of the network that is needed for the distributed computing system to operate. To assess the state of the global network, the state of local network contributes invaluable information (Chandy, 1985). The state of a processor is a function of a numerous factors such as local memory, stacks, and so on (Kshemkalyani, 2011).

States of Distributed Computing System

This section describes a quantitative method to measure the level of state in a distributed computing system. Suppose a particular computing process is running, which can be denoted as p_i after the occurrence of an event $e_{i,x}$ where the next event is denoted as $e_{i,x+1}$ (Kshemkalyani, 2011). The level of state is a function of the process and event. Hence, the level of state is defined as $l_{i,x}$. The occurrence of an event is executed when a signal is sent out. In other words, the sending and receival of a signal allows us the quantify the level of state of a processor. The volume of information contained between the event of sending the signal and receiving the signal is exactly the content of the level of state. This leads to a more intriguing discussion of the past and the future of an event (Mattern, 1988).

Past and Future of a Distributed System Event

The previous section discussed the literature to measure the level of state of an event that goes through a distributed system (Kshemkalyani, 2011). This section continues to dive into the

realm of past and future discussion that evolves around the concept of an event in a distributed computing system (Mattern, 1988).

A Curious Application

Block-chain and cryptocurrency is one of the most important inventions in the 21st century. The heart of a block-chain system is a concept called Ledger which is created to support double-entry referencing or bookkeeping (Herlihy, 2019). It is established to preserve the encoding information for decipher for cryptocurrency (Herlihy, 2019). In laymen's term, the ledger stores the information of a transaction. For instance, John has provided a coin to Mary. The action of the coin leaving John's account and entering Mary's account is stored in ledger. Since the information marks exactly what has happened, the information in ledger can be extremely valuable when the number of transaction amount is significantly large. This means that the ledger needs to be encrypted and needs to be free of tampering or other malicious activities. During this process, the ledger, with the transaction information stored inside, is encrypted onto a document. The document is then passed into a pool of ledgers which then checked by an algorithm called a "black-box protocol" one by one. It may sound clumsy, but the benefit is that such algorithm is universal, and it can easily compartmentalize the consensus protocol (Herlihy, 2019). What is this magical algorithmic protocol? This answer to this question leads to the concept of a cryptographic hash function.

Cryptographic Hash Function

Any cryptographic algorithm has a hash function to encrypt the information that needs to be secret (Herlihy, 2019). In other words, it is stated that a hash function is often denoted by $H(\cdot)$. For any value v, the rule to create the hash function $H(\cdot)$ is that the function is (1) easy to reproduce, and (2) it is not possible that there exists v' such that $H(v) = H(v')$. Hence, there

cannot be any two values that pointing to the same output (Herlihy, 2019). Because many crypto miners are mining these algorithms, this nature gave birth to many hash functions. Though there are many possible ways to do this, the solutions are not infinite. When two miners are going for the same block, the miners may stack or append the blocks together to continue to work on the project jointly. This action is called fork (Herlihy, 2019). Though many researchers have provided novel methods, the common way is to work on the block that is the longest (Sompolinsky, 2016) (Herlihy, 2019). The probability a block, after it is designed in a block-chain, will be replaced by a new block after miners work on it decreases exponentially (Garay, 2015) (Pass, 2017). Hence, the uncertainty that if a ledger can exist permanently is always nonzero. That is, we can never be truly sure that the ledger stores information precisely for an infinitely amount of time. This nature creates many problems for the design of block-chain technology and the field.

Conclusion

The article investigated the procedure of developing multithreading techniques. The paper started with literature review of the general trend of developing multi-threaded cores in the field of parallel computing. In addition, the paper provided some analysis of the architecture and the mathematical form of how performance is measured. Then the paper discussed and landed on a review of an application of how machine learning applications are affected using different chips for computation candidate.

References

Birman, K. P. (1993). The process group approach to reliable distributed computing. *Communications of the ACM*, 37-53.

Chandy, K. M. (1985). Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 63-75.

Gargees, R. S. (2020). Multi-stage distributed computing for big data: Evaluating connective topologies. *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, 0626-0633.

Herlihy, M. (2019). Blockchains from a distributed computing perspective. *Communications of the ACM*, 78-85.

Kshemkalyani, A. D. (2011). Distributed computing: principles, algorithms, and systems. *Cambridge University Press*.

Mattern, F. (1988). Virtual time and global states of distributed systems. *Univ., Department of Computer Science.*, 215-266.

Prasad, S. G. (2015). Topics in parallel and distributed computing. *Morgan Kaufmann*.

Waldo, J. W. (1996). A note on distributed computing. *International Workshop on Mobile Object Systems*, 49-64.

Recommend a Fault Tolerant Program

Yiqiao Yin

Computer Science, PhD

TIM8121 Distributed Algorithms and Parallel Computing

Professor William Souza

September 2022

Recommend a Fault Tolerant Program

The task in this assignment is to devise a faulty program tolerance program. The proposed setup is based on the same scenario as described in previous assignment. The scenario states that a computer system is in place for a dual-core Central Processing Unit (CPU). The goal is to invert the colors of an image. The image is sized 150 KB and the image file is in bitmap format. There is a RAM in place, and it is operating as a character array. While the previous develop a multi-core algorithm using C++ to invert the colors in parallel, the current task is to design a dual-core processor to ensure the faulty tolerance can be identified. There is much previous research in this field. The setup analysis of processor performance has attracted a lot of interests in the literature. This research heavily focused on the performance of dual core or multi-core processor architectures (Ojeyinka, 2015). Their work is valuable for developing the Intel processor architecture over time where most buyers were buying computers on the spot when walking in the mall or shopping area. The performance system is extremely necessary for a consumer and not many tests are verified and publicly available (Russell, 1999) (Reinders, 2007) (Ojeyinka, 2015). To quantify the analysis of multi-core processor, it is important to note that the performance is largely dependent on a variety of different factors such as design architecture, experimental design, operating systems, different types of compilers, implementation of software program, and so on (Ojeyinka, 2015).

Task

In this homework assignment, the desired application program requires an interface of using OpenMP. The system is required to support multi-platform as well as shared-memory multiprocessing programming languages. The common languages in this field within the scope of this assignment is C, C++, and Fortran. The system needs to operate on many different

platforms with instructions set for architectures to run and operate on different computing systems such as Solaris, AIX, HP-UX, Linux, and MacOS.

OpenMP: Open Multi-Processing

The Open Multi-Processing or OpenMP is an application programming interface that helps scientists to develop stack of code to support multiprocessing using C++ or other related languages. The system and pipeline are independent from Message Passing Interface or MPI which is a standard and portable message-passing design for parallel computing (Gropp, 1996). This provides the coding and programming standards since 1996 and it certainly defined the major roles of parallel computing. Furthermore, this platform is backed by major players in the industry such as IBM. The platform also supports multiple different computer programming languages such as Fortran, C, and C++. The task is to define a software algorithm to investigate the BMP file format or the bit map image file. The next section describes the process of the instructions for this algorithm to operate.

Served Image File

This section reviews the digital format delivered for the algorithm designed for this task. The most popular digital format that stores the images operates around a double for-loop or nested for loop. This was originally designed by Microsoft (Kakugawa, 2000). The formal name is called a device-independent bitmap or DIB. It is a format that is used to create the definition of bitmaps in a various of color resolutions (Kakugawa, 2000). The file structure consists of fixed-size definitions with rows and columns of pixels being well defined within the range of image frame. Once defined, the pixels forma arrays and the arrays form a slice of greyscale of scheme taking brightness level from a color category. The information, described by pixel levels, is then saved by byte sized image file and there can be different byte sizes (Bourke, 1998).

Instructions

The major component is to design an algorithmic compiler that works with shared memory machines or in parallel. In addition, it can also be extended to support Graphical Processing Units (GPU). Before we start introducing the algorithm, let us introduce the arguments in the algorithm.

Definition of Arguments

The definition of arguments can be tricky. One needs to note that in C++ the arguments are defined using specified data types. The argument of a C++ program is regarding to the values that are passed into a function when the function is called upon. The data type of the arguments is defined upstream in program unless downstream the data type is redefined. Usually, the common practice is to define the data types in the beginning of the code and the data types remain the same throughout the code to be consistent. First, a function to return the maximum number of threads need to be defined. The function should be able to support OpenMP platform and it produce an integer output.

```cpp
#pragma omp parallel
int omp_get_max_threads()
```

The function directly displays an integer of the number of threads used in the multi-core processor. It is generally set by a variable named “number of threads” or similar sorts. In addition, we should also check the number threads in the current execution. This can be done using the following function.

```cpp
int omp_get_num_threads()
```

The function above displays the number of threads that is currently used in the operation. It generally defaulted to be one, because most workload starts with one CPU. Another function that needs to be used is the function that returns thread ID so that the workload can be set in parallel if needed.

```cpp
int omp_get_thread_num()
```

This function checks the total number of threads in the processor and then assign a number to be a particular core for potential usage or distribution. This number or index of number is also important for naming convention in case a program requires to distribute jobs to different cores. This function allows the scientists to not overload all the workload to one thread and to execute the jobs more efficiently systematically.

Each image is assumed to have a fixed width and height. These concepts are denoted as $width$ and $height$. These two arguments define the size and the byte-size of the image file. The number of pixels for a two-dimensional array that stores the greyscale or brightness of a color scheme forms a single slice of image with size $width \times height$. The basic colors of computer science are red, green, and blue. Hence, a colored image would have to have three two-dimensional arrays. The total number of pixels would be $width \times height \times 3$. Each pixel would take values of greyscale or also known as the brightness level from 0 to 255 where 255 is the brightest and 0 is the darkest. The brightness level or the greyscale can be used on a color scheme. At a particular index of row and column, there are three pixel-values where each pixel value refers to a particular color scheme, i.e., Red, Green, and Blue, respectively.

Algorithmic Design

The algorithm design for converting the image has many different versions. The naïve version is to check every pixel and invert the number of the values from black to white and vice versa all on one core. This is the slowest and the algorithm can simply be a nested for-loop. There are three for-loops. The first for-loop investigates every element of the $width$. The second for-loop investigates every element of the $height$. The third for-loop investigates every color. Hence, the space to search through each pixel is precisely $width \times height \times 3$.

For this assignment, the program utilizes a dual-core processor to evaluate the faulty tolerance. In order to execute this task, the operation requires execution on the entire image array data and the process occurs in each core. Then the algorithm makes comparisons. The comparisons take two arrays together to evaluate their identicality. If the condition fails to check and they are not identical, the loop is repeated, and the process continues until the output arrays are the same. The workflow is demonstrated in Figure 1. Multi-core Processor Design.

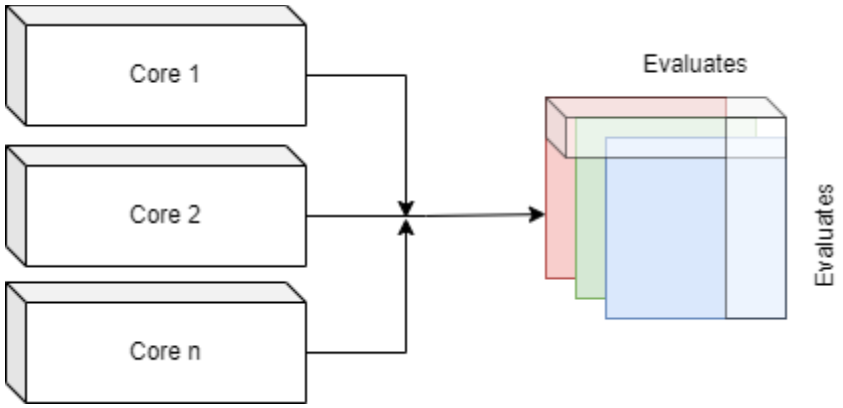

Figure 1. Multi-core Processor Design

A "Hello World" Example

The algorithm starts with a "hello world" example to guide readers through using OpenMP. The code can be found in the following.

```cpp
```

```
#include <omp.h> // this is required for this algorithm
#include <stdio.h>
int main(int argc, char *argv[]){
    printf("OpenMP running with %d threads\n", omp_get_max_threads());
    #pragma omp parallel {
        // code below executes by all threads
        printf("Hello World from thread %d\n", omp_get_thread_num());
    }
    return 0;
}
```

The code uses the function to check the maximum number of threads that is currently running and the threads that are used to print the “hello world” statement.

Code Walk-through

The algorithm (presented in “Supplement” section) starts with the definition of a helper function. The helper function identifies the bit map image and records the data from the image.

```
    ifstream _to_find_it("this_bit_map_image.txt");
    char ch;
    int i, c=0, sp=0;
    while(_to_find_it) {
        _to_find_it.get(ch);
        i=ch;
        if(i >=48 && i =< 57)
        c++;
    }
```

This is a helper function so there is no output generated yet. The function above uses a while-loop to register the entries from the image file. Then it counts the number of integers within a certain range. The current range is selected to be from 48 to 57. The count is checked by adding the variable *c* by one every step in the while-loop. The main algorithm starts with checking the helper function and by making sure the image file is open.

```
    ifstream my_file_ ("this_bit_map_image.txt");
    if (my_file_.is_open()) {…}
```

The function above sets the steps up for the main while-loop to execute in the algorithm. Then the algorithm reads in the character by using the helper function and uses a local variable ch to register the entry of the image file. This entry is then used as information to define the $loop$ which is an array-like data structure. The algorithm checks the entry inside the loop. There is a if-statement or a conditional statement in place to verify the information. The code can be shown below.

```
if(array[loop]==48) {
    array[loop]=49;
}
else if(array[loop]==49) {
    array[loop]=48;
}
```

The outcome of the code replaces the entries inside of the object $loop$. If it is “1”, we replace it with “0”. If it is “0”, we replace it by “1”.

Conclusion

This task for this week focuses on the design of a dual-core processor system to check the faulty tolerance in the system. The paper proposes a design of the algorithm and code snippet is presented in the supplement section.

Bibliography

Bourke, P. (1998). Bmp image format. *BMP Files*.

Gropp, W. L. (1996). A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing*, 789-828.

Kakugawa, H. (2000). A Device-Independent DVI Interpreter Library for Various Output Devices. *Proceedings of the TUG 2000 Conference*, 102-107.

Ojeyinka, T. O. (2015). Performance Analysis of Dual Core, Core 2 Duo and Core i3 Intel Processor. *International Journal of Computer Applications*.

Reinders, J. (2007). Intel threading building blocks: outfitting C++ for multi-core processor parallelism. *O'Reilly Media, Inc.*

Russell, M. K. (1999). Testing on computers: A follow-up study comparing performance on computer and on paper. *Boston College*.

Supplement

Please see the code snippet below.

```
#include<iostream>
#include<fstream>
using namespace std;
int main() {
    ifstream _to_find_it("this_bit_map_image.txt");
    char ch;
    int i, c=0, sp=0, ub=48, lb=57;
    while(_to_find_it) {
        _to_find_it.get(ch);
        i=ch;
        if(i >= ub && i =< lb)
        c++;
    }

    int array[i];
    short loop=0;
    ifstream my_file_ ("this_bit_map_image.txt");
    if (my_file_.is_open()) {
        while (! my_file_.eof() ) {
            _to_find_it.get(ch);
            array[loop] = ch;
            if(array[loop]==0) {
                array[loop]=1;
            }
            else if(array[loop]==1) {
                array[loop]=0;
            }
            loop++;
        }
        my_file_.close();
    }

    else cout << "can't open the file";
    system("PAUSE");
    return 0;
}
```

Comparing Types of Parallelism

Yiqiao Yin

Computer Science, PhD

TIM8121 Distributed Algorithms and Parallel Computing

Professor William Souza

September 2022

Comparing Types of Parallelism

This assignment investigates the background literature of the parallelism. The main component of this paper evaluates different types of parallelism for efficiency. The types of parallelism investigated in this work are bit-level parallelism, instruction-level parallelism, multicore parallelism, and distributed computing. The paper is organized in the following. First, the paper reviews the literature and discusses the background of related work. Next, the paper discusses a list of concepts required to solve the problems and tasks in this assignment. The third section presents analysis of different scenarios of parallelism. Moreover, the paper expands the answer and discusses potential extreme cases related to the tasks. Last, the paper makes a conclusion summarizing the analysis and comparisons.

Literature Review

Modern day parallel computing aims to increase the efficiency of workflow in computer science. Under today's paradigm, the Big Data Era has led to many machine learning and deep learning models that required iterative computing methods which raised the stakes of the parallel computing (Dahl, 2011) (Hinton, 2012) (Cireşan, 2010) (Martens, 2010) (Zhang, 2014) (Deng, 2012). It is important to understand different types of parallelism and hence be able to make sound comparisons. This allows computer scientists to devise feasible plan to multiple different programs and write appropriate algorithms that fit for the applications. In addition, parallelism is one of the main concepts in information technology particularly in the field of Artificial Intelligence (Nagy, 2006). This paper has proposed and- and or-parallelism (Nagy, 2006). This leads to a whole generation of growing market share of the Central Processing Unit (CPU) and Graphical Processing Unit (GPU) market embedded by using controllers such as microchips (Düll, 2015).

Key Concepts in Computation Efficiency

This section introduces the notions that are used in the Proposed Answer and Analysis section in order to ensure that the unit of analysis used in metrics are up to standard.

32-bit Computing

In computer science specifically in the sub-field of computer architecture design, the 32-bit integers are data units that come with length 32-bit wide. In other words, the data stream can store information up to 2^{32} different values, i.e., $2^{32} = 4{,}294{,}967{,}296$ in length. Depending on the representation of the usage of integers, the range of the solutions that can be applied to are in 32-bit of format. Moreover, the size of the running memory is capped at 4GB. This means that a processor that has 32-bit memory can at most operate with 4GB in its running memory. The data transferring is another side of the story to understand the meaning of a 32-bit machine. The concept of 32-bit on data transfer refers to the capacity of moving 32 bits of information per running or per clock cycle. In laymen's term, it is the amount of information going through the machine every time the processor performs a computer operation.

64-bit Computing

With 32-bit being discussed in previous subsections, the concept of 64-bit machine can be understood more easily. It is the information size that gets to be transferred when the processor executes the operation. This is the core component of the supercomputer. The evolution from 32-bit computing power to 64-bit computing power allowed humanity to visualize the exponential growth available to handle much larger computation workload. This growth is presented in Figure 1. Growth of Supercomputers, measured by the unit of analysis FLOPS. A supercomputer is a computer that is integrated with many cores or processors in a way that can handle much larger workload than a personal laptop or desktop. The performance of

supercomputers is measured by floating point operations per second or FLOPS. In Figure 1. Growth of Supercomputers, the x-axis is measured by time (in year) and the y-axis is measured by FLOPS. For the fields of subjects in Science, Technology, Engineer, Mathematics, and Medicine (STEMM), the supercomputers were introduced as early as 1960s (Hoffman, 1949).

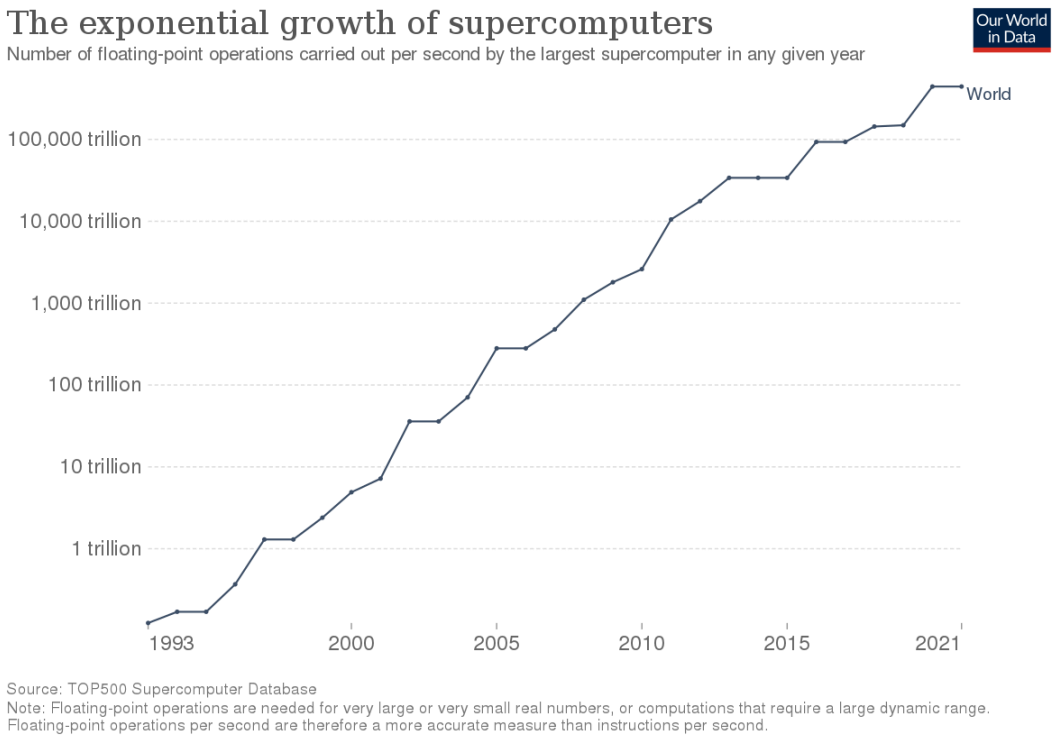

Figure 1. Growth of Supercomputers

The evolution of this exponential growth was led by the IBM 7030 Stretch (Smotherman, 2010). The original was carried out by a small team led by John Griffith in Poughkeepsie, New York before they were told wrong by Ralph Palmer (Bell, 2011). It was a shame because the design could have been the blueprint of point-contact transistors or surface-barrier transistors (Bell, 2011). The point-contact transistor is the first ever transistor introduced by the Bell Lab with the functionality of amplifying electrical signals (Hoddeson, 1981).

3.5 GHz

The implementation of the Internet of Things and 5G Era that we are all enjoying today is largely due to the evolution of 3.5 GHz range. The 3.5GHz forms the center core of the 5G technology and it provides the basis for the implementation of high-speed internet.

Survey Different Types of Parallelism

This section surveys four different types of parallelism in detail. The four types are: bit-level parallelism or BLP, instruction-level parallelism or ILP, task-level parallelism or TLP, or distributed-level parallelism or DLP.

Bit-level Parallelism

For large-scale computing from 1970 to 1986, the enhancement of supercomputer is largely due to the usage of doubling the computer word size or, in other words, the number of texts the computer can process for every operation cycle (Culler, 1999). The reason for the increase of computation efficiency is based on the following fact. The increase of text size allowed in the data processing and data transferring results in reduction of workload in instructions.

The concept of bit-level parallelism is presented in Figure 2. Bit-level Parallelism. For example, a word could have a size of 16-bit. Suppose it is desired for computer scientists to process this 16-bit word into a processor. At a bit-level, the word is required to have processing power of 16-bit. If the processor is 8-bit, then two cores are required to get the job done. However, one easy way to execute bit-level concept is to increase the size of the processor at a bit level. Hence, a solution is to use a 16-bit processor to conduct the workload. The execution cycle would need to only run once while the 8-bit processor would need to be stacked and ran twice.

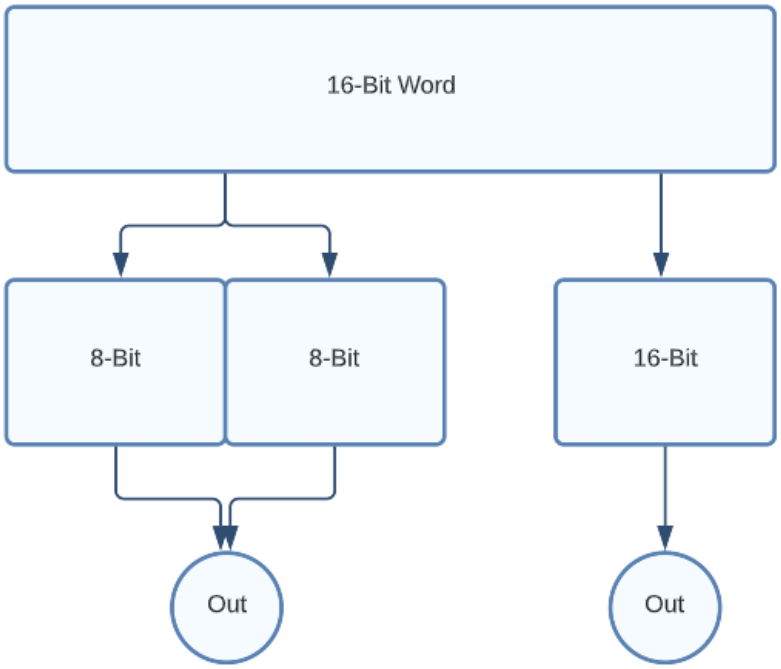

Figure 2. Bit-level Parallelism

Instruction-level Parallelism

For large-scale computation, instruction-level parallelism can be used to achieve higher efficiency. The concept of instruction-level parallelism relies on the nature that an instruction is issued per computation cycle (Culler, 1999). This provides the crucial key required to design instruction-level parallelism. As one of the major evolutionary advancements in the 1980s and 1990s, instruction-level parallelism has been the major force to drive parallel computation (Culler, 1999).

The concept of instruction-level parallelism is presented in Figure 3. Instruction-level Parallelism. As the name suggests, the instruction-level parallelism is completely dependent on the instruction by the program which is designed by the computer scientists. For example, suppose an additive series $1 + 2 + 3 + 4 + \cdots + 10$ is required to be computed. The naïve solution is to manually do the addition, which can be computationally costly. However, a mathematician could work out a "smart" way. For example, denote S_1 to be $1 + 2 + 3 + \cdots + 10$

and denote S_2 to be $10 + 9 + 8 + \cdots + 1$. The series $S_1 + S_2$ would then be $11 + 11 + 11 + \cdots + 11$ and there should be a total ten 11s. Hence, the result of $S_1 + S_2$ is $11 \times 10 = 110$. Notice that the two series, S_1 and S_2, are the same. Hence, to figure out the sum of one series, the math can simply be $\frac{110}{2} = 55$, which is the correct results. Hence, at an instruction-level, the first instruction can be $11 \times 10 = 110$ and the second instruction can be $\frac{110}{2} = 55$, which would complete the workload.

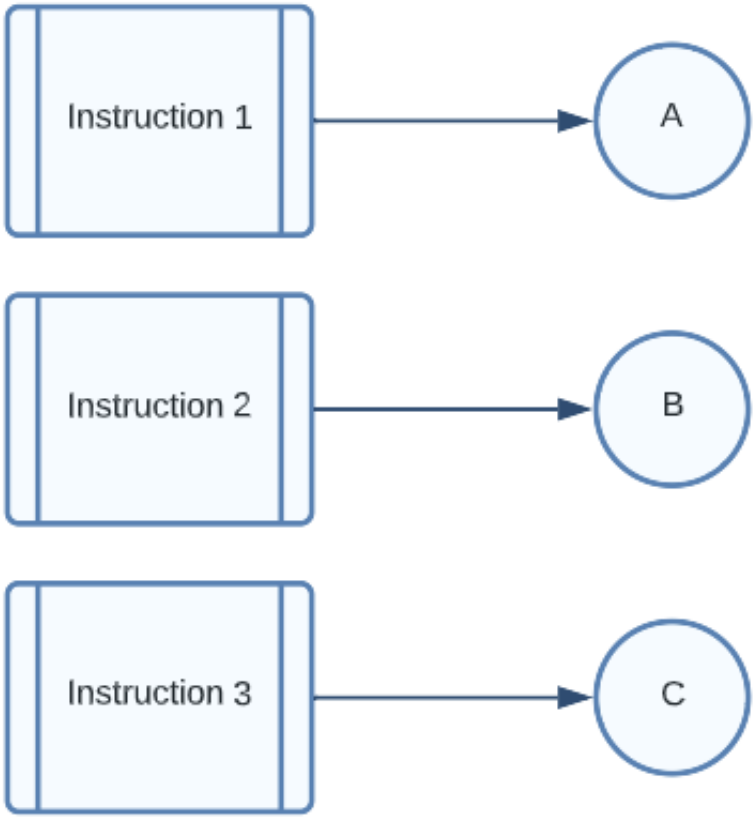

Figure 3. Instruction-level Parallelism

Multicore Parallelism

As the third parallelism to be introduced, the multicore parallelism is rather easier to understand then the previous two candidates. The multicore parallelism is to execute the workflow in parallel utilizing more than one core simultaneous (Zheng, 2014). The cost would

be to acquire the multicore processors or even multi-processors. The prices of the Nvidia Graphical Processing Units (GPUs) are trending higher than before. The Radeon RX 6400 costs $137 in USD while its advanced version after two generations have prices raised to $204 in USD. The latest GeForce GPUs are the same as well. The initial introduction of GeForce RTX 3050 was priced at $260 in USD. Its predecessor GeForce RTX 3080 is priced at $585 in USD. All numbers are cited according to Tom's Hardware (Hardware, 2022).

One crucial caveat, though not necessarily a weakness, is the premise of parallelism. If an algorithm is sequentially executed, the multicore parallelism would not be valid. For example, many machine learning algorithms rely on the famous gradient descent algorithm. Gradient descent algorithms use a pre-defined loss function as the object function to compute gradient, which is a partial derivative of the loss function with respect to different parameters. The parameters are iteratively updated until the algorithm meets the desired error threshold. Gradient descent algorithm is sequentially executed. In other words, it cannot be made faster by having multiple different computer chips. The reason is that a step of parameter update can only happen after the previous step is done. However, algorithms can be designed to fit for the parallelism premise. For another example, bootstrap is an algorithm that samples the data many times allowing repeated data entries to construct confidence intervals. Each sample is considered independent from another. Hence, bootstrap algorithm can be paralleled.

To demonstrate the concept of parallelism, the following example can be used Figure 4. Simulation of Pi. Suppose a computer scientist desires to use simulation to recover the value of π. The area of a circle can be used $Area = \frac{1}{4}r^2$ and hence the area of a quarter of a circle is well defined. If the area of a circle is given, the value of π can be reverse engineered. The following simulation sample data from the uniform distribution with lower bound to be 0 and upper bound

to be 1, $x \sim U(0,1)$. The sampling of 100 is independent from the sampling of 1000 data points, and hence can be paralleled. Every sample provides a certain number of data points with x-axis and y-axis both in the range of $(0,1)$. These data points have coordinates allow us to compute the distance it has towards the origin, $(0,0)$. The data points with distance towards the origin less than one can be coded into the red group otherwise they are coded as the black group. The number of red data points should form a quarter of a circle as the sample size increases. Hence, the recovered π values should get closer and closer to the theoretical value which is $\pi = 3.1415921653$ ... This trend can be seen using the plot on the right in Figure 4. Simulation of Pi.

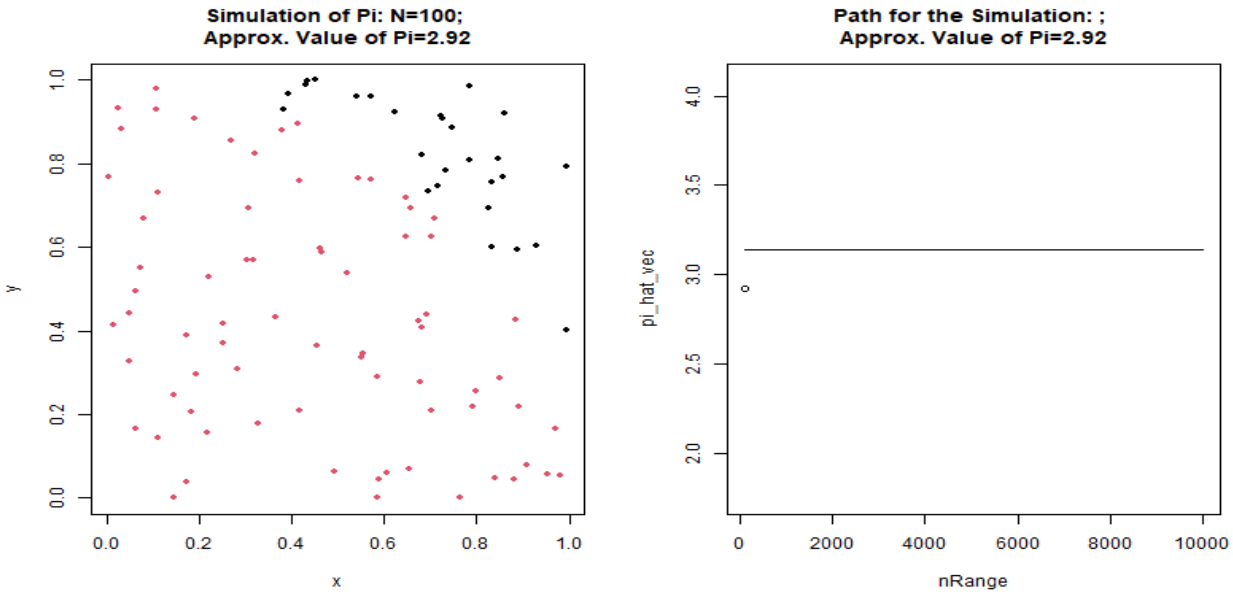

Figure 4. Simulation of Pi

Distributed Parallelism

Like the instruction-level parallelism, the distributed parallelism also shares a common goal amongst the workloads. In other words, the distributed parallelism system is a network of processors that execute jobs using a unified goal given by the computer scientist. In laymen's term, the terms "parallel computing" and "concurrent computing" have also been used a lot. To provide clarification amongst different terminologies, in this article we refer to all these terms

interchangeable with the concept of "distributed parallelism" (Peleg, 2000). The only difference between parallel computing and distributed computing is that in parallel computing the memory is shared across all processors whereas in distributed computing each processor has its own dedicated memory (Papadimitriou, 1994).

Technical Overview

This section discusses the technical component required for the proposed answer and analysis in the next section.

The parallel distribution formula is the first concept to be introduced. Suppose the render time for each data file is t_r and the number of data file to be rendered is represented by f. Then the total amount of parallel computing time, t_p, is introduced by Equation 1. Parallel Distribution Equation.

Equation 1. Parallel Distribution Equation

$$t_p = t_r \times f$$

Next, we introduce the distributive distribution formula. The formula is created to compute the total time spent on the distributed system, t_d, using the rendered time for each data file and the rendered time for multiple data files using distribution system. Suppose the render time for each data file is denoted as t_r, the same as it was denoted as in Equation 1. Parallel Distribution Equation. Also suppose the file size is s, the transfer speed is t_t. Then the distributed distribution equation is defined as Equation 2. Distributed Distribution Equation.

Equation 2. Distributed Distribution Equation

$$t_d = t_t + \left(\frac{s}{t_t}\right) \times (f + 1)$$

Notice that the second term in the Equation 2. Distributed Distribution Equation has a "+1" term inside. This is because the data files are sent to the system all together but transferred back out to

the system as output one by one. In addition, the second is a product of s/t_t and $f + 1$, because the formula is computing the total number of rendering time for using the distributed system.

Proposed Answer and Analysis

This section provides the answer to the questions raised in each of the four scenarios. In addition, analysis is also provided to elaborate each answer. The discussion is carried out according to each of the scenarios in the following manner. For each of the discussions, a description of the tasks is provided. Then the analysis of the work is provided to elaborate on the premises. Last, the answers are provided to address the results of the computation for each discussion.

Scenario 1: 64-bit 3.5 GHz single-core CPU

The scenario of this task states that a database of 10 videos is collected. The videos are 1GB each and would be required to be rendered. Suppose that it takes 128 seconds to render each file using a 32-bit 3.5 GHz single-core CPU. In other words, for a 32-bit 3.5 GHz single-core CPU to finish 10 files, it would take 1280 seconds. We can use this number as a starting point. If the processors are 64-bit instead, the speed would have doubled and hence the time would now be $\frac{1280}{2} = 640$ seconds.

Scenario 2: 32-bit 3.5 GHz single-core CPU with instruction-level pipelining

In a separate scenario, how long does it take for a 32-bit 3.5 GHz single core CPU to finish the task with pipelining. The workload of instruction pipelining would increase the time spent on executing this task. Hence, the total time would be the 128 seconds plus whatever it takes to finishing the pipelining. The pipelining is constructed using the stages. If there are 10 files, the assumption is that the steps are taken sequentially. If each step requires 1 second, it is

expected to increase 10 seconds. Hence, the entire task would take $128 + (10 + 1) = 139$ seconds.

Scenario 3: 32-bit 3.5 GHz quad-core CPU without pipelining

Without instruction level pipelining and the increment of 64-bit core enhancement, the computation speed would be the same as the original design. If there are 10 files and each file takes 128 seconds, then it is expected to take $128 \times 10 = 1280$ seconds in total to finish the task.

Scenario 4: Five 32-bit 3.5 GHz single-core CPUs online with 100 Mbps downloading

Last, suppose there are five 32-bit 3.5 GHz single-core CPUs in the network design. The network is connected to the internet with a downloading speed of 100 Mbps. For a file that is 1 GB or 1000 MB in size, this translates to 10 MB per second. In addition, there are 10 files, so the expected time added is $10 \times (10 + 1) = 110$ seconds. In total, the program is expected to finish at $128 + 110 = 238$ seconds.

Conclusion

This capstone project answers the computation questions following the proposed quantitative methods. To do this, this paper starts with Literature Review and extends the survey to different key concepts to understand computation efficiency. In addition, this paper surveys different types of parallelism which include Bit-level Parallelism, Instruction-level Parallelism, Multicore Parallelism, and Distributed Parallelism. The article finally delivers the Proposed Answer and Analysis section to address the premises and solutions.

References

Bell, G. (2011). Out of a closet: the early years of the computer museum. *Dependable and Historic Computing*, 130-146.

Cireşan, D. C. (2010). Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, 3207-3220.

Culler, D. S. (1999). Parallel computer architecture: a hardware/software approach. *Gulf Professional Publishing*.

Dahl, G. E. (2011). Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on audio, speech, and language processing*, 30-42.

Deng, L. Y. (2012). Scalable stacking and learning for building deep architectures. *2012 IEEE International conference on Acoustics, speech and signal processing (ICASSP)*, 2133-2136.

Düll, M. H. (2015). High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Designs, Codes and Cryptography*, 493-514.

Hardware, T. (2022). GPU Prices 2022.

Hinton, G. D. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 82-97.

Hoddeson, L. (1981). The discovery of the point-contact transistor. *Historical Studies in the Physical Sciences*, 41-76.

Hoffman, A. R. (1949). Supercomputers: directions in technology and applications. *No. 04; e-book.*

Martens, J. (2010). Deep learning via hessian-free optimization. *ICML*, 735-742.

Nagy, B. (2006). On the notion of parallelism in artificial and computational intelligence. *Proc. of HUCI*, 533-541.

Papadimitriou, C. H. (1994). Computational Complexity. *Approximation and Complexity*.

Peleg, D. (2000). Distributed computing: a locality-sensitive approach. *Society for Industrial and Applied Mathematics*.

Smotherman, M. &. (2010). IBM's single-processor supercomputer efforts. *Communications of the ACM*, 28-30.

Zhang, K. &. (2014). Large-scale deep belief nets with mapreduce. *IEEE Access*, 395-403.

Zheng, W. T. (2014). Fast databases with fast durability and recovery through multicore parallelism. *11th USENIX Symposium on Operating Systems Design and Implementation*, 465-477.