Accelerated Exact K-medoids via Parallel Computing.

MSc Computer Science

Supervisor: Professor Max Little

Ben McGuirk

ID: 2158143

School of Computer Science

College of Engineering and Physical Sciences

University of Birmingham

2023-24

Abstract

Cluster analysis is a longstanding technique used in machine learning for several important applications such as bioinformatics and image processing. One such example of cluster analysis is the K-medoids problem, where a novel, exact and tractable algorithm was recently developed. This new algorithm, named EKM, can guarantee global optimality in polynomial time, with respect to the number of medoids (K). For research areas like early cancer diagnosis or extreme weather prediction, speed and accuracy are vital, positioning EKM as a significant algorithmic development. This study aimed to investigate the impact of parallel computing on the runtime of EKM's combination generator. To determine this, the Python Multiprocessing module was implemented, as well as two multi-core CPU architectures. We here demonstrate that parallel multiprocessing provides speedup to the EKM generator. In addition, core scaling provided further acceleration to the generator. Further investigation is warranted to determine the optimal utilisation and coordination of the Multiprocessing module, as well as multi-core CPU architecture, depending on the size of each use case. Restructuring of the EKM algorithm would enable similar investigation of a parallel GPU implementation of EKM, for applications where the necessary hardware is available.

Acknowledgements

I would like to thank my supervisor, Professor Max Little, for his feedback and support during my project. This includes providing me with comments on my drafts as well as constructive feedback in each meeting.

I would also like to extend my thanks to my project inspector, Professor Mark Lee, for his guidance and feedback on my project demonstration. This was particularly useful for gaining confidence in my figures and overall project direction.

Abbreviations

AI	Artificial Intelligence
ALU	Arithmetic Logic Units
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
BnB	Branch and Bound
CLARANS	Clustering Large Applications based on RANdomized Search
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DNN	Deep Neural Network
DVFS	Dynamic Voltage Frequency Scaling
EUVL	Extreme Ultraviolet Lithography
EX	(Instruction) Execution
FC	Fully Connected
FPGA	Field Programmable Gate Array
GIL	Global Interpreter Lock
GPU	Graphics Processing Unit
HDL	Hardware Description Languages
HPC	High Performance Computing
ID	Instruction Decode
IF	Instruction Fetch
ILP	Instruction Level Parallelism
ISA	Instruction Set Architecture
LAN	Local Area Network
LLC	Last Level Cache
MEM	Memory Access
MIMD	Multiple Instructions Multiple Data
MIP	Mixed Integer Program
MISD	Multiple Instructions Single Data
ML	Machine Learning
PAM	Partitioning Around Medoids
PIM	Processing-In-Memory
RNN	Recurrent Neural Network
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
TPU	Tensor Processing Unit
WB	Write Back

Contents

1 Introduction and Background

The adoption of parallel computing in the late 20th and 21st centuries has been a key development for many of the technologies we take for granted today. In a data-driven world, the demand for sophisticated parallel architecture has never been higher. Despite significant progress, the full potential of parallel computing in machine learning (ML) is yet to be realised, with many more algorithms showing potential for parallelisation. One such algorithm is the K-medoids clustering problem, where a novel, exact and tractable algorithm was developed recently by Professor Max Little and Xi He, named EKM [1]. This is the first known tractable (polynomial time complexity) and optimally exact solution to the K-medoids clustering problem. The authors highlight its suitability for parallelisation because no communication is required between processes, making it embarrassingly parallel. Parallelisation will provide significant reductions in the overall running time of the new algorithm, further improving its efficacy. A novel algorithm that is both optimal and tractable has great potential for critical applications of ML where confidence and accuracy are vital. As ML algorithms evolve and transform, there will continue to be opportunities for optimisation through parallel computing.

1.1 Sequential Computing

Historically, computer software has been written for serial execution. In serial computing, algorithms are executed as a single stream of instructions, processed by a single central processing unit (CPU), one at a time [2].

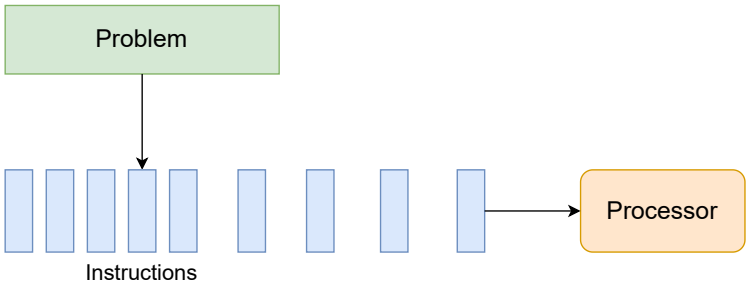


Figure 1: Serial computing example showing instructions being executed by the processor one at a time.

The limitations of serial computing are easily inferred, however great efforts were made in the 20th century to maximise the performance of serial computing. Despite initial improvements, performance began to plateau towards the end of the century [3]. Two main limitations to be elucidated are frequency scaling and memory bottlenecks.

1.1.1 Frequency Scaling

Frequency scaling, or dynamic voltage frequency scaling (DVFS) is a technique used in high-performance computing (HPC) to adjust clock frequency. It has previously been a key factor in improved computer performance. The time it takes for a program to execute is calculated by multiplying the number of instructions by the average execution time per instruction. The clock frequency of a processor indicates the number of cycles it can execute per second, therefore increases in clock frequency results in lower execution time per instruction and thus faster overall program execution. Despite this benefit, frequency scaling is bidirectional. In some scenarios, decreasing frequency scaling is preferable to conserve energy while demand is low [4]. One prominent example of frequency scaling is NVIDIA Graphics Processing Unit (GPU) Boost, a technique that dynamically adjusts clock speeds based on GPU temperature and workload [5].

Increases in clock frequency leads to higher voltages and subsequent higher power consumption and heat generation in the processors. In severe cases thermal throttling can occur when the processor has to lower clock frequency to avoid overheating. A key turning point was the cancellation of Intel's Tejas and Jayhawk processors due to their power consumption. [6] The benefits of decreasing processor frequency have also plateaued, mainly due to larger static power consumption [7]. This was a defining moment as frequency scaling was generally no longer seen as a pivotal technique for HPC [6].

Overall, pushing the limits of frequency scaling is complex, and requires advanced coordination between software and hardware. As well as this, not all applications obtain the same benefits of frequency scaling, and latency caused by changing frequencies can also impact performance [4].

1.1.2 Memory Bottlenecks

In serial computing, the speed of memory access has not been able to keep up with the increasing speed of CPUs, leading to increases in memory latency. Memory bandwidth is the amount of data that can be transferred in a given period between the CPU and memory. It is also restricted, which severely impacts performance for processing large amounts of data or carrying out memory intensive tasks [4]. These memory restrictions will get worse as architectures are scaled, which is a key technique for the development of ML models.

Modern solutions for these memory bottlenecks include the use of multi-level cache memory and faster structures that keep data closer to the processor. Although cache memory is much faster than main memory, the demands of copying and moving data remain. In contrast to the standard processor-centric computing model, some researchers have proposed memory-centric models consisting of combined memory/logic chips [8]. These chips provide much higher bandwidth and lower latency. The issue, however, with memory-centric computing

is that they are more challenging to program because you need to anticipate future model compression methods. Furthermore, coordinated implementation of processing-in-memory (PIM) requires support for new instruction set architectures (ISAs) [9].

Serial computers require less hardware than parallel architectures, however, this comes at the expense of performance. The remaining problems of overheating and power-intensive processors, as well as memory and scalability constraints, have led to wider adoption and increased emphasis on parallel architectures.

1.2 Parallel Computing

Parallel computing is a type of computation where multiple instructions can be processed at the same time. This is most useful for tasks involving lots of calculations and data, such as training and evaluating ML models. Different types of parallel computing include data, bit-level, instruction-level, and task parallelisation [2].

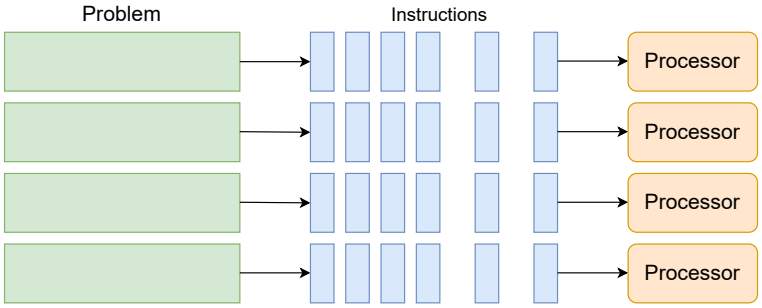


Figure 2: Parallel computing example showing instructions executed in parallel by four different processors. This diagram highlights an immediate speedup compared to figure 1 for embarrassingly parallel problems where no inter-processor communication takes place.

1.2.1 Processor Scaling

After discovering the complications of frequency scaling, manufacturers began to develop multi-core CPUs. In a multi-core CPU each core is independent from each other, however, they have synchronous memory access, allowing for fluent coordination on tasks. One derivation of scaling predictions such as Moore's law (see section 1.2.7) is that every 2 years the number of cores per processor will double, however in practice it can be seen that this has not transpired. The standard number of cores per processor has gone from 4 in 2012, to around 4-16

presently [10].

In an ideal scenario, the relationship between the number of cores and speedup would be linear, however, speedup is limited by the percentage of code that is parallelisable. This concept is formalised by Amdahl's law:

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

Where:

- $S(n)$ is the theoretical speedup of the execution of the whole task;
- n is the number of processors;
- p is the proportion of parallelisable code.

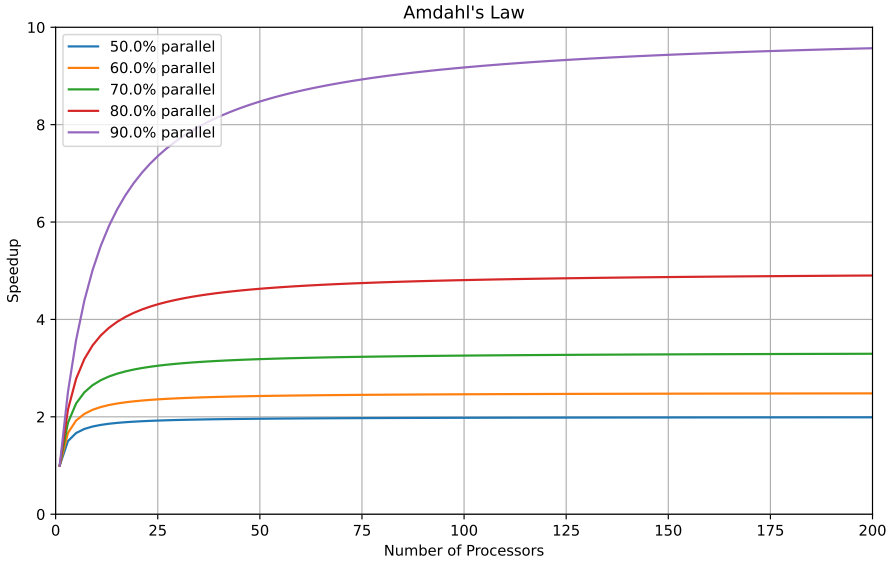


Figure 3: Graphical representation of Amdahl's law, showing the diminishing returns of increasing the number of processors, even with a high proportion of parallelisable code.

Amdhal's law shows that if 90% of a program can be parallelised, then the maximum speedup that can be achieved is a factor of 10, even with an infinite number of cores [11]. The constraints imposed by Amdhal's law are highly restrictive, however, they are only applicable when the problem size is fixed. One trend in ML is the rapid scaling of models and training datasets,

and this expansion can be represented more accurately by Gustafson's law [12]. Gustafson's law gives a more realistic prediction of parallel performance when the problem size can be increased. This is because with Amdahl's law, the total amount of work is fixed when adding more processors, whereas work per processor remains constant when adding more processors with Gustafson's law enabled by increasing problem size. Gustafson's law is defined as:

$$S(n) = n - p(n-1) \tag{2}$$

Where:

- $S(n)$ is the speedup factor;
- n is the number of processors;
- p is the proportion of **non**-parallelisable code.

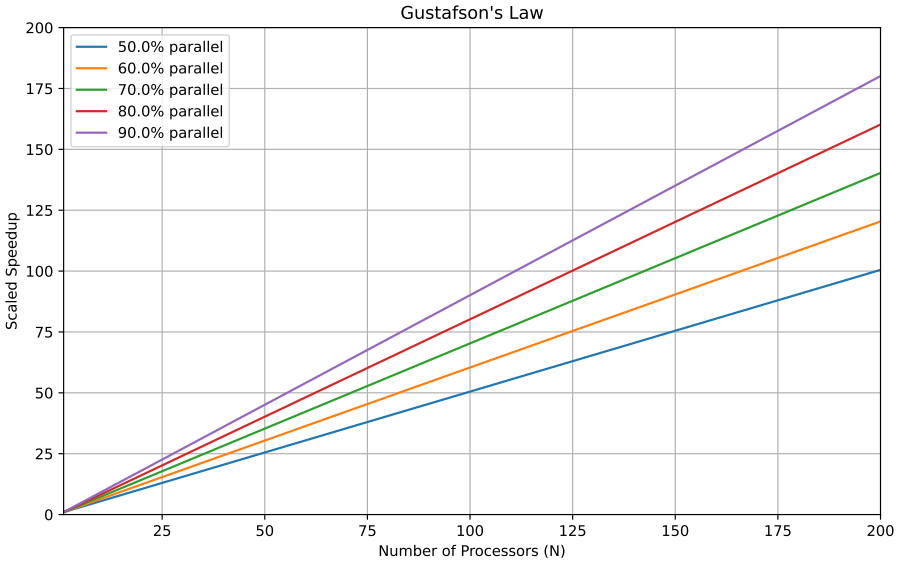


Figure 4: Graphical representation of Gustafson's law, showing a linear representation of the performance of multiple processors.

1.2.2 Data Dependencies

A key factor in the practical implementation of parallel algorithms is understanding data dependencies. The maximum speed of an algorithm is dependent on the longest chain of dependent calculations, otherwise referred to as the critical path. This dependency can be seen in Amdahl's law represented by the

proportion of code that cannot be parallelised. Most ML algorithms and their training datasets are not dependent on long chains of calculations, so there is a lot of potential for parallelisation [13]. As mentioned previously, the new EKM algorithm by Max Little and Xi He involves no processor communication, making it highly suitable for parallelisation [1].

Parallel program subtasks, known as threads, need concurrent access to objects in memory, for example when performing updates on a variable. If two threads perform operations on shared memory without proper synchronisation, it can lead to critical data errors. Mutual exclusion must be provided for each thread via a lock. A lock will allow one thread exclusive access to the shared object and prevent other threads from reading or writing it. This ensures the correct execution of all threads. The section of each thread that requires mutual exclusion is called the critical section [14].

Programs can be grouped according to their degree of parallelism. Programs that need their subtasks to communicate many times a second, are classed as fine-grained. Programs that do not need their subtasks to communicate many times a second are classed as coarse-grained, and programs that never need their subtasks to communicate are classed, as mentioned, as embarrassingly parallel [15].

Despite the numerous benefits of parallel computing, caution and careful consideration must be taken regarding the tasks most appropriate for parallel processing, as less complex tasks can perform worse due to added communication time between cores and threads. When the overhead from communication is greater than the time saved through parallelisation, adding more cores or threads will negatively impact performance. This is known as parallel slowdown [16, 17].

1.2.3 Flynn's Taxonomy

Flynn's Taxonomy is one of the earliest classification systems for parallel computing, and is still widely used today for designing modern processors. The system classifies architectures based on the number of concurrent instructions and data streams it can handle. The four main categories are SISD (single instruction stream, single data stream), SIMD (single instruction stream, multiple data streams), MISD (multiple instruction streams, single data stream) and MIMD (multiple instruction streams, multiple data streams). SISD is best for simple, sequential tasks where there may be risk of parallel slowdown. SIMD is suitable for tasks involving large amounts of data and repeated calculations. An example of this in EKM is the efficient, recursive combination generator presented [1]. This is also a prime example of Bellman's *principle of optimality* [18]. The principle of optimality determines that if you can break down a problem into smaller subproblems, the solutions to these subproblems can be combined to solve the overall problem. MISD is an uncommon architecture, mainly used for fault tolerance. MIMD is suitable for a range of tasks and can be executed

with architectures such as distributed systems [19].

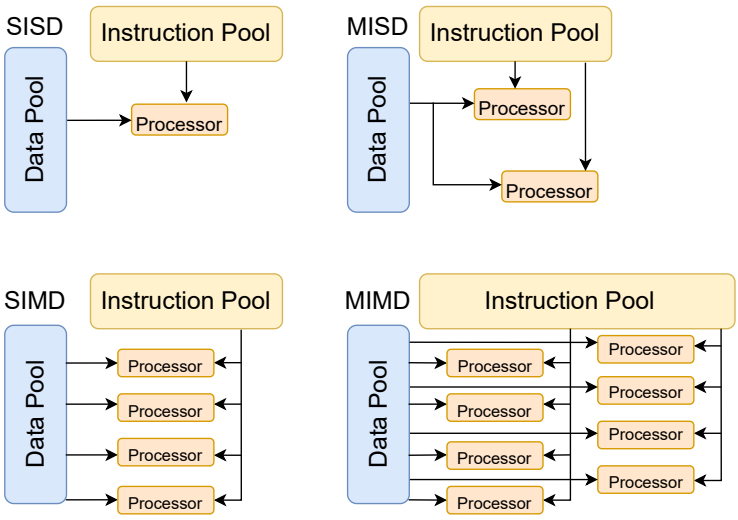


Figure 5: The four main categories of Flynn's Taxonomy: Single Instruction Single Data, Single Instruction Multiple Data, Multiple Instructions Single Data and Multiple Instructions Multiple Data.

1.2.4 Granularity

Bit-level parallelism refers to the technique of increasing processor word size. Increasing processor word size reduces the number of operations needed to execute an instruction. For example, a 16-bit processor adding two 32-bit integers will take multiple operations, whereas execution on a 32-bit processor will take just one operation. This is because the 16-bit processor has to first sum the lower 16 bits of both numbers, then sum the upper 16 bits of both numbers, whilst accounting for any carry. The first electronic computers were single bit, and increases in word size trended from here for many years up until the early 2000s which saw the widespread adoption of 64-bit processors [4].

One modern bit-level principle commonly seen in machine learning is bit-level sparsity parallelism. Bit-level sparsity is the presence of lots of zeros in binary representation of data. For example the integer 8's 8-bit binary representation is '00001000'. The presence of bit-level sparsity can be used to one's advantage by compressing the data efficiently. Efficient compression can be achieved by specialized hardware like GPUs that can be instructed to skip zeros or process multiple bits in parallel. In machine learning, ubiquitous matrix operations can be accelerated through bit-level sparsity parallelism. Furthermore, neural networks often have sparsity in their weights and activations which can

also be exploited to improve training and inference speeds [20].

Two main methods in deep learning for increasing weight and activation sparsity are pruning and quantization [21]. Pruning works by setting less important elements of the model to zero. This is done by setting a value threshold, below which parameters may be set to zero. This increases the sparsity of the model, which can then be exploited via previously mentioned methods, to accelerate training and inference. Quantization, alternatively, is about reducing the precision of the parameters of the model. Floating point numbers are converted to lower bit-width integers such as 8-bit representation, improving memory usage and computational efficiency. 'Deep compression', a three-stage pipeline introduced by Han et al in 2016, is a compression method that reduced the storage requirements of AlexNet [22] 35× whilst maintaining accuracy. The three stages consisted of pruning, quantization and Huffman coding [21].

Emphasis has also been placed on other types of parallelism such as instruction-level parallelism (ILP). ILP refers to the technique of increasing the number of operations that can be performed simultaneously. Without ILP, processors can only execute one instruction per clock cycle. These are referred to as sub-scalar processors. The two main types of ILP are software and hardware. The hardware approach implements dynamic parallelism, where the instructions to be run in parallel are decided at run time. The software approach implements static parallelism, where the instructions to be run in parallel are chosen by the compiler [4].

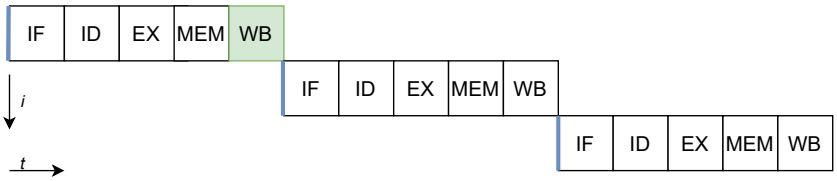


Figure 6: Instruction execution with no instruction-level parallelism. The five stages shown here are instruction fetch (IF), instruction decode (ID), instruction execution (EX), memory access (MEM) and write back (WB). The downward arrow i represents each downward step as a new 5-stage sequence of instructions. The three blue lines represent the start of each instruction sequence. The arrow t represents the direction in which each sequence is executed. The green square indicates which instruction is being executed at the current time step, in this instance, just the WB instruction of the first sequence.

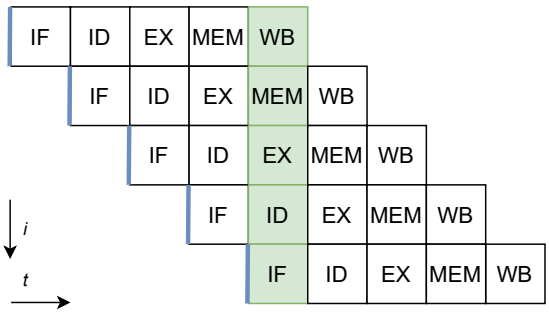


Figure 7: Instruction execution with instruction-level parallelism, showing the accelerated performance provided by pipelining. In the given example, in one clock cycle five instructions are being executed (green squares) compared to just one without pipelining in figure 6.

1.2.5 Hardware

Memory in parallel computing comes in the form of shared or distributed memory. Shared memory is an architecture where many processors access the same physical memory space. Distributed memory is an architecture where each processor has its own memory space, giving rise to inter-processor communication. Shared memory systems are simpler to program and more suitable for smaller amounts of parallelism. Distributed memory systems offer better performance for large amounts of parallelism whilst being more complex to implement. Despite these differences, modern HPC models often take a hybrid approach, for example, distributed clusters (distributed memory) of shared memory multicore processors [4]. These two architectures offer different capabilities and are suited to different use cases, so strong prior knowledge of the types of tasks and data being processed is key for parallel computing performance.

Parallel computing architectures use caches, which provide a small and fast memory structure. To keep track of values and ensure consistency across the system, a cache coherency system is often used [4]. Cache coherency ensures that shared data that ends up in multiple caches is represented consistently. For example, if there are multiple cached copies of a data item, and one copy is updated, the cache coherency system will update all copies of the cached data item. This is often achieved through a method called bus snooping [23].

Parallel computers can be categorised according to the level of support the hardware provides for parallelisation. The main categories commonly seen today are distributed computing, cluster computing, massively parallel computing and grid computing. Distributed computing involves multiple computers, often in different locations, connected and working on a unified problem. The comput-

ers communicate via a network and each uses its own processors and memory. This makes distributed computing highly scalable, and a powerful technique for applications such as ML model training [24]. Cluster computing involves a group of closely connected computers that work on a unified problem as a single system. Rather than needing to communicate over a network, the computers in a cluster are usually connected via a local area network (LAN) [25]. Massively parallel computing involves the scaling of thousands of processors to work on different parts of a problem synchronously [4]. Grid computing is a collaborative form of distributed computing often involving a large-scale network. In grid computing resources are often shared amongst multiple organisations [26].

1.2.6 Software

To capitalise fully on the advances seen in parallel computing hardware, there have been sufficient advances in software to support their integration into practical problems like ML. Developments in software for parallel computing include parallel programming languages, libraries and application programming interfaces (APIs). These are often categorised based on the memory architecture they are designed for (shared, distributed or hybrid).

Due to the rise and wide adoption of GPU programming, the role of compute kernels has become vital for parallel computing performance. Compute kernels are programs that are optimised for accelerators like GPUs. When using GPUs, kernel instructions can be executed in parallel by multiple threads. One common implementation is in CUDA (Compute Unified Device Architecture), a parallel computing platform developed by NVIDIA to maximise the capabilities of their GPUs [27].

As ML models and parallel infrastructure scales, failures occur more often due to increased complexity. In these scenarios application checkpointing is often deployed to minimise wasted time, by keeping a record of resource allocations and object states. This means that if a failure occurs, the system can be restored from its most recent checkpoint rather than the very beginning of the program. For systems with a large number of parallel processors, application checkpointing is a key technique for performance [28].

1.2.7 Moore's Law

In 1965 Gordon Moore observed that in an integrated circuit, the number of transistors doubles on average every year [29]. Later, in 1975, this was revised to doubling every two years. Current ideas on Moore's law and its future are split, with prominent figures in the technology world divided by opinion. Historically there have been multiple predictions of Moore's law coming to an end, however, it has continued to be aligned with industry progress [3].

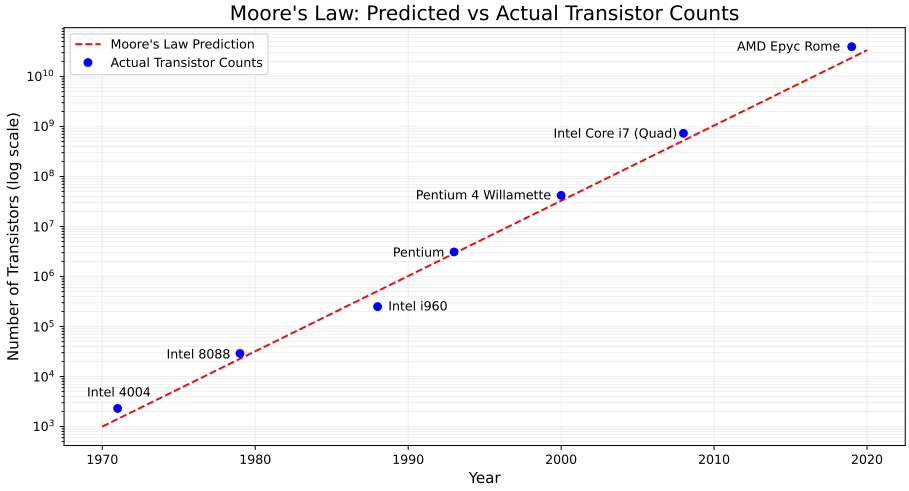


Figure 8: Showing the persistence of Moore's law from the 1970s up until 2020, with some specific historical examples. Key reasons for its endurance are the limitations of sequential computing that led to the development and wider use of multi-core CPUs and GPUs. These include the plateaus of frequency scaling, typical power and thus overall single-thread performance. More examples of industry chip transistor numbers can be seen in Max Roser et al's 2023 article [30].

Dennard scaling formalises the relationship between transistor size and power use. Named after Robert Dennard, this scaling law states that as transistor size decreases, power density remains the same. This therefore means that as transistor size decreases, power remains proportional with area. This implies that the number of transistors can be doubled, without increasing the amount of power consumed [31]. A key event that led to scepticism towards Moore's law was in 2005 when Dennard scaling began to break down, however, transistor numbers continued to increase as seen in figure 8.

Moore's second law, also known as 'Rock's law' after Arthur Rock, states that every four years, the cost of a semiconductor chip fabrication plant doubles. A key contributor to this growing cost is extreme ultraviolet lithography (EUVL), which is a technique used in high-volume integrated circuit manufacturing. EUVL uses extreme ultraviolet light to create detailed patterns on thin slices of semiconductors known as silicon wafers [32]. Although there is a clear relationship between Moore's law and Rock's law, it is thought that the fabrication plant cost per transistor would be a more accurate constraint on Moore's law [33].

Despite the numerous sceptics and pessimistic predictions, the semiconductor industry is passionate about keeping Moore's law alive and often uses it

as a benchmark for progress. A recent example was announced in 2022 by Intel research, who want to keep Moore's law on course by producing a trillion transistor package by 2030. This is partly made possible by a novel material developed by intel that is just three atoms thick [34].

1.3 Artificial Intelligence Accelerators

Artificial Intelligence (AI) accelerators, or deep learning processors, are specialised systems or hardware made specifically for ML tasks. These tasks can be seen in domains such as robotics, autonomous vehicles and computer-aided diagnosis [35].

1.3.1 Field Programmable Gate Arrays

Field programmable gate arrays (FPGAs) are integrated circuits that can be configured and programmed to perform specific functions. FPGAs are made up of logic blocks that can perform complex operations or act as simple logic gates. A key feature of FPGAs is their ability to be reprogrammed, unlike application-specific integrated circuits (ASICs), allowing for more flexibility with regard to the number of different tasks that can be performed. As well as logic blocks, FPGAs consist of programmable interconnects, input/output pads and memory elements [36, 37].

Whilst providing flexibility and high performance, FPGA programming requires very specialised skills. This is because FPGAs make use of Hardware Description Languages (HDLs), which are very difficult to convert to from a higher level programming language [38].

1.3.2 Tensor Processing Units

Tensor Processing Units (TPUs) are AI accelerators developed by Google in 2016. An example of ASIC architecture, TPUs are specialised for deep learning tasks like matrix multiplication [39, 40]. Early research from Google claims that TPUs are 15-30 times faster than CPUs and GPUs whilst being 30-80 times more energy efficient [41]. This new chip consists of matrix multiply units, which improved the time complexity of matrix multiplication from $O(n^3)$ to $O(n)$. This is because instructions in TPUs can operate on tensors of data, as opposed to one piece of data per instruction. High performance of matrix multiplication is also facilitated by high bandwidth memory, which provides accelerated processing of large-scale operations and alleviates memory constraints that are ubiquitous in ML computation [41]. Applications of TPUs are widespread, with many research areas benefiting from their performance. Examples in healthcare include medical natural language processing and genomics research on protein folding [42, 43].

1.3.3 Graphics Processing Units

Graphics Processing Units (GPUs) are specialised circuits originally developed for gaming applications. GPUs are suited to SIMD tasks (see section 1.2.3) as they can contain thousands of cores for efficient parallel processing. SIMD allows GPUs to carry out the same instruction on many data points synchronously. On top of this, memory bandwidth is commonly higher in GPUs than in CPUs, allowing for faster data transfer. Matrix multiplications are a common operation in 3D video rendering, which gave rise to the efficacy of GPUs in gaming. This was leveraged during the breakthroughs of deep learning, where the demand for sophisticated hardware increased to handle these operations [44]. GPU programming has been made accessible by platforms like CUDA and OpenCL [27]. Apart from gaming and machine learning, GPU programming is also prominent in cryptocurrency mining and scientific computing.

A 2019 study on DL training, by Wang et al, found that TPUs are well-suited for convolutional neural networks (CNNs) and recurrent neural networks (RNNs), whilst GPUs are preferable for fully connected (FC) networks. They also found that CPUs were advantageous for the largest FC networks [45]. A key issue highlighted by the authors is a lack of rigorous benchmarking prior to their study, with previous efforts giving misleading conclusions. Transformer models train 3.5× faster on TPUs than on GPUs [46]. The issue with this assertion, is that memory bandwidth bottlenecks occur on TPUs for FC networks of more than 4k nodes. This is a key factor for their suitability to CNNs because of its weight sharing, and GPUs suitability to FC networks. They further argue that the largest FC networks are suitable for CPU implementation due to the need for model parallelism on GPUs and TPUs. While thorough proof is provided for their analysis, the study could be improved by testing the performance of inference, multi-node systems and accuracy of each processor type. The study compared Google's cloud TPU v2/v3 with NVIDIA's V100 GPU, and Intel's Skylake CPU platform [45].

2 *K*-medoids Clustering

K-medoids is an unsupervised learning problem in the sub-category of clustering algorithms, alongside similar problems such as K-means and K-median clustering [47, 48]. The general blueprint for clustering algorithms is to be able to partition a given dataset into K-number of clusters in a D-dimensional space. This is important for a number of varied applications such as email spam classification, geographical crime analysis and diagnostic systems.

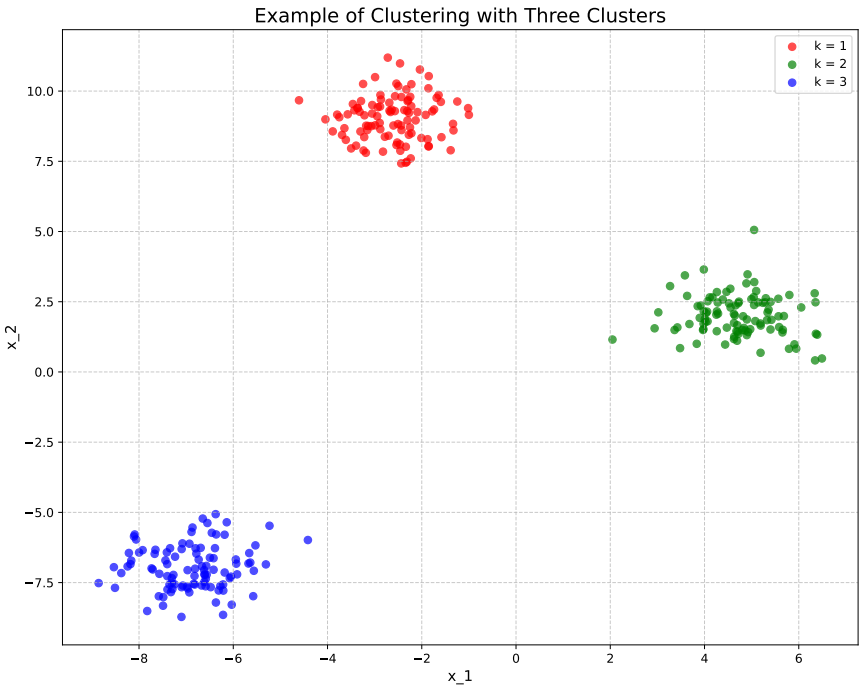


Figure 9: K-clustering example visualisation with 3 clusters in 2 dimensions.

The loss or cost function of clustering is derived from a chosen distance measure between each data point and its assigned centroid. Often the squared Euclidean distance is used, which between two points can be defined as:

$$\text{dist}(x, \mu) = \sum_{d=1}^{D} (x_d - \mu_d)^2 \tag{3}$$

Where:

- x, μ are two points in Euclidean d-dimensional space;
- x_d, μ_d are coordinates at dimension d;

- D represents D-dimensional space.

With this distance function, we can derive the overall cost of a particular configuration for K-clustering by taking the sum of distances for each centroid and their assignments:

$$E_{\text{K-clustering}} = \sum_{k=1}^{K} \sum_{n=1}^{N} \text{dist}(x_n^k, \mu_k) \tag{4}$$

Where:

- E is the total loss;
- K is the total number of clusters;
- N is the total number of assignments per cluster;
- μ_k is the centroid for cluster k;
- x_n^k is the n^{th} data point assigned to cluster k.

From here the optimal configuration of assignments z and associated centroids μ can be found with the configuration that minimizes the objective function $E_{\text{K-clustering}}$:

$$(\hat{z}, \hat{\mu}) = \arg\min_{z} E_{\text{K-clustering}}(z, \mu) \tag{5}$$

The difference between these clustering algorithms is the type of centroids they cluster around. Aptly named, K-means clustering calculates each centroid position by taking the mean of all points assigned to it. Respectively, this is also the case with K-median clustering and the median of all assigned points to each cluster. K-medoids takes a slightly different approach by assigning an actual data point as each centroid (often referred to as medoids). K-medoids originates from a 1990 book by Kaufman et al [49], where their partitioning around medoids (PAM) algorithm was presented. A medoid of a cluster can be seen as the most centrally located point, proven by having the smallest sum of distances between all other points in the cluster. The PAM algorithm implements a greedy search for these medoids, which may not find the globally optimal solution, however, it is much faster than an exhaustive search, which often leads to combinatorial explosion.

In more detail, the original PAM algorithm works in two stages; build and swap. In the build stage medoids are initialized by selecting K inputs from the dataset. In the swap stage all pairs of medoid and non-medoid data points are evaluated iteratively. If swapping the medoid and non-medoid lowers cost, the swap is performed. This is iterated until no improvement can be made, at which

point the model has converged. In being greedy, the model may find the global minimum (optimal solution), or it may converge on a local minimum. This property, shared by other heuristic solutions to K-medoids, can be explained by the fact that locally optimal choices taken at each step only consider the immediate best option. An early optimal step may prevent the algorithm from finding the globally optimal solution, as it converges or 'gets stuck' in a local minimum.

An exact solution to the K-medoids problem has previously been NP-hard, meaning that prior to EKM [1] there was no known algorithm that could solve K-medoids optimally in polynomial time, for any possible inputs. As such, many heuristic solutions have previously been proposed, whilst any exact solutions have lacked applicability to large datasets due to their intractability.

2.1 Exact K-medoids

In May 2024, Xi He and Max Little presented the first known exact and tractable algorithm for the K-medoids problem, with worst-case $O(N^{K+1})$ time complexity [1]. Heuristic solutions to the K-medoids problem are computationally efficient and can be used for large datasets, however, they are not able to ensure retrieval of the globally optimal solution. While this trade-off may be attractive in some cases, the consequences of not finding an exact solution can be significant, especially for sensitive applications like healthcare or extreme weather prediction.

Previous research has provided few exact algorithms, with Branch and Bound (BnB) being one of the most popular [50]. He and Little outline three main problems with efforts up to this point. The first is that these algorithms have a computation time limit, which means that exact solutions are often not calculated, especially for large N or K. The second is that transparency of worst-case complexity is often not provided, restricting the algorithm's practical application. Finally, mathematical proofs are often missing, making it impossible to ensure that the globally optimal solution is always identified.

2.1.1 Theory

EKM is derived by leveraging *Bird-Meertens formalism* [51] to develop an efficient and accurate algorithm. By default, the K-clustering problem can be solved by producing all possible combinations of centroids, as one of them must be the optimal configuration that solves (5). The issue, as stated earlier, is that exhaustive enumerations lead to combinatorial explosion. One benefit of K-medoids is that the centroids are actual data points (medoids), and so the number of arguments to be evaluated for (5) is much smaller compared to K-means and K-median clustering.

The authors explain that since this exhaustive approach is provably correct, any algorithm derived through correct equational reasoning steps is also provably correct. Firstly, the K-medoids problem is formalised as a mixed-integer program (MIP), meaning it involves both integer and continuous variables. The algorithm was formally tested with the squared Euclidean distance function, however, any objective function can be used when calculating configuration cost. This provides flexibility for future implementations whilst maintaining polynomial complexity and global optimality.

A common method for solving combinatorial MIPs is the *generate-evaluate-select* algorithm:

$$s^* = sel_E(eval_E(gen(D))) \tag{6}$$

Starting with the generator function *gen* all possible combinatorial configurations are enumerated. For this problem, configurations are lists of data points representing each medoid, and the search space is the set of all possible combinations of these medoids. The evaluator function *eval* calculates the cost of each configuration generated by *gen*. Finally, the selector function *sel* selects the best configuration s^* with respect to the cost. (6) is a brute-force search, so the exact solution will always be found. As stated, any algorithm derived through correct equational reasoning steps will also be exact [1].

To transform this brute-force search into a tractable solution, a transformational programming principle called *shortcut fusion* is applied. In combinatorial optimisation, many problems like the generator function can be as efficient as recursion with linear complexity $O(n)$. Recursive implementation allows you to *fuse* the evaluator directly into a recursive generator. This means that configurations can be generated and selected through a single recursive program. This implementation is underpinned by Bellman's *principle of optimality* described previously [18]. Within this principle, the overall problem is to enumerate all possible combinations S^{n+m} for list $l_1 \cup l_2$, and the subproblems are the enumeration of all possible combinations S^n, S^m for lists l_1, l_2. To produce these combinations efficiently, the cross-join operator $\circ$ can be used as part of a convolutional product $conv(\circ, l_1, l_2, k)$ for lists l_1 and l_2. For example [[1], [2]] $\circ$ [[3], [4]] = [[1,3], [1,4], [2,3], [2,4]]. Therefore, a recursive combination generator can be defined as:

$$\begin{aligned} gen_{combs}(0, k) &= merge([], k) \\ gen_{combs}(n, k) &= merge(merge([x_n], k), gen_{combs}(n-1, k), k) \end{aligned} \tag{7}$$

Pattern matching allows the merge function to be called with different input cases:

$$
\begin{aligned}
&merge_E([], k) = [[[]]] \\
&merge_E([x_n], k) = [[[]], [[x_n]]] \\
&merge_E(l_1, l_2, k) = conv(\circ, l_1, l_2, k)
\end{aligned}
\tag{8}
$$

The final efficient recursive generator is presented as $gen_{combs}(n, k)$ for data sequence D_i by $S^n_{\leq k}(D_i)$. For example, $gen_{combs}(3, 2) = S^3_{\leq 2}([x_1, x_2, x_3]) = [[[]], [[x_1], [x_2], [x_3]], [[x_1, x_2], [x_1, x_3], [x_2, x_3]]]$. At this stage of the paper, the algorithm is rendered as:

$$
s^* = sel_E(eval_E(gen_{combs}(n, k))) \tag{9}
$$

From here, the evaluator function is fusible with the generator function because the objective function E is split up into a sum of loss (one loss value per data item). This means that the objective function can be calculated cumulatively alongside generator recursion. This idea is implemented for $eval_E$ and gen_{combs} by defining:

$$
\begin{aligned}
&evalgen_{E,combs}(0, k) = merge_E([], k) \\
&evalgen_{E,combs}(n, k) = merge_E(merge_E([x_n], k), evalgen_{E,combs}(n-1, k), k)
\end{aligned}
\tag{10}
$$

where $merge_E$ is defined as

$$
\begin{aligned}
&merge_E([], k) = [[([], \infty)]] \\
&merge_E([x_n], k) = [[([], \infty)], [([x_n], \infty)]] \\
&merge_E(l_1, l_2, k) = conv(\circ_E, l_1, l_2, k)
\end{aligned}
\tag{11}
$$

Where ∞ is the initial objective value. The convolution function with cross join operator mentioned previously is also improved to provide evaluation updates ($\circ_E$). This intermediate fusion is formalised as:

$$
\begin{aligned}
s^* &= sel_E(eval_E(gen_{combs}(n, k))) \\
&= sel_E(evalgen_{E,combs}(n, k))
\end{aligned}
\tag{12}
$$

This synchronous generation and evaluation allows storage of the best current configuration on each recursive stage. This facilitates the fusion of the selector function into the synchronous $evalgen_{E,combs}$ with only partial configurations being stored. This is the final derivation that formalises He and Little's EKM algorithm [1]:

$$
\begin{aligned}
s^* &= sel_E(evalgen_{E,combs}(N,K)) \\
&= selevalgen_{E,combs}(n,k) \\
&= EKM(n,k)
\end{aligned} \tag{13}
$$

To recap, performing the EKM algorithm for clustering length N data inputs into K clusters, the overall complexity is $O(N^{K+1})$. The authors reiterate its guarantee as an exact algorithm due to its correct derivation through shortcut fusion transformations.

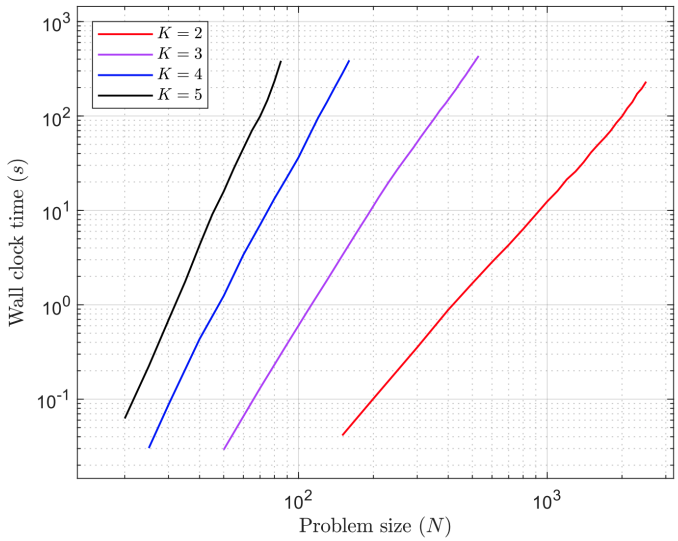
2.1.2 Performance and Evaluation

The performance of EKM is analysed and compared to a number of approximate algorithms across 22 different synthetic and real-world datasets. These algorithms include PAM [49], Fast-PAM and Clustering Large Applications based on RANdomized Search (CLARANS) [52]. Fast-PAM builds on the original algorithm presented by Kaufman et al [49], and provides an $O(K)$ speedup in the 'swap' phase of PAM mentioned previously. This is achieved by implementing additional 'swaps' in each iteration, allowing the algorithm to converge faster and provide a practical solution for large datasets [52]. In the first step of the CLARANS algorithm, the search space is visualised as a high dimensional hypergraph. This is a structure where edges can connect any number of vertices, rather than just two. With an edge representing a 'swap' of medoid and non-medoid, the algorithm performs a randomized greedy search. The CLARANS algorithm also benefits from the 'swap' phase modifications made to obtain the Fast-PAM algorithm.

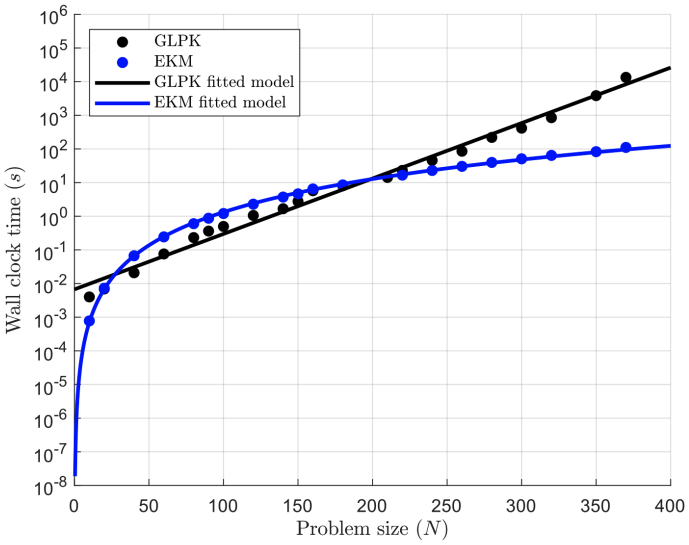
EKM is also compared to a recent implementation of BnB, one of the most popular exact algorithms up to this point [53]. BnB is an efficient algorithm design paradigm for solving optimisation problems [50]. The algorithm enumerates configurations by partitioning the search space into smaller sets (branching), then setting upper and lower bounds to estimate the optimal solution in each subset (bounding). Upper and lower bounds provide benchmarks for the objective function (4) in order to solve (5). The 2022 implementation of BnB by Ren et al [53], uses a Lagrangian relaxation method to provide a lower bound at each BnB node. One issue with Lagrangian relaxation, however, is it is not guaranteed to find the globally optimal solution. This is addressed by the authors, with their lower bounding method providing an exact solution as it branches only on the region of the medoids. Overall, this means that the search space and optimality gap (the gap between lower and upper bounds) are significantly reduced.

It was found that against the aforementioned algorithms, EKM always produced the best objective values. Interestingly, for several datasets tested, PAM and Fast-PAM also found the globally optimal solution. In addition, the authors

acknowledge that for use cases where efficiency takes precedence over accuracy, these heuristic algorithms may be more desirable [1]. This also applies to problems with large K, as the EKM algorithm ($O(N^{K+1})$) becomes intractable.



(a)



(b)

Figure 10: (a) EKM time complexity tested with log-log wall clock run time (seconds) and number of clusters K ranging from 2-5. Shows polynomial time complexity with respect to K as predicted (appear as linear functions on log-log plot) [1]. (b) Comparative plot between EKM and BnB MIP solver (GLPK) [54] with $K = 3$. Because of the log-linear scale, exponential run time (GLPK) plots as a linear function, and polynomial run time (EKM) plots as a logarithmic

function. These figures originate from Max Little and Xi He's EKM paper, used here with permission from the authors [1].

Throughout, the authors describe EKM's suitability for efficient parallelisation due to its embarrassingly parallel structure, meaning there is no dependency or communication between processes. Particular reference is made to the generator described in section 2.1.1, which has an inherently parallel structure. Acceleration through parallel processing would further reduce the algorithm's run time and significantly improve its performance for critical use cases.

2.2 Aim

This project aims to focus on the acceleration of the EKM combination generator. Specifically, parallel computing techniques will be investigated and evaluated for their potential to speed up the generator, and thus the EKM algorithm as a whole.

3 Methodology

For the parallel implementation of EKM, a multi-core CPU system was chosen over a multi-CPU system for a number of reasons. In a multi-core system, each core is on the same chip which means that latency is significantly reduced when communicating between each core due to their shorter distances from each other [55]. Whilst there is no inter-process communication in EKM, the multi-core design still reduces latency for returning the results of each process compared to each process being on a separate chip. Each core has an individual cache system allowing for fast memory access. On top of this, last level cache (LLC) and memory controllers are shared resources between each core that facilitate latency reduction. Lastly, multi-core systems are more energy efficient than multiple-chip systems which enables higher clock frequency (see section 1.1.1) [56].

In section 3.3 the original EKM generator is illustrated and explained further. This is followed by an intermediate implementation in section 3.4 where the input list is split in two, with each half being passed into the *gen* function (7) one after the other (sequentially). This is referred to throughout as the 'split-sequential generator'. In section 3.5 the Multiprocessing module is outlined, before being used to develop a parallel EKM generator in section 3.6. This is referred to as 'parallel generator 1'. The techniques applied to develop the first parallel generator are then taken further in section 3.7, with the development of a second parallel generator where a larger proportion of the generator is executed in parallel. This is referred to throughout as 'parallel generator 2'. Both parallel generators use the Multiprocessing module to parallelise the EKM *gen* function (7), where the focus is on efficiently enumerating all possible combinations of an input list N, up to combinations of size K (number of desired medoids).

3.1 Legal, Social, Ethical and Professional Issues

Legal issues include intellectual property rights for any algorithm used as well as compliance with software licensing agreements for any modules used. Social issues include accessibility for a wide range of individuals or organisations and the impact of technological development on employment. Ethical issues include the potential misuse of an algorithm for harmful use cases as well as gender and racial bias in machine learning datasets. Professional issues include proper credit attribution and accurate documentation of research.

3.2 Project Management

The main project management technique used was a Kanban board, with the Notion app. This has been used throughout the project giving a clear timeline as well as goals to be achieved at each stage. At a higher level of abstraction, goals included completion of individual sections of the report e.g. to complete a first draft of the results section by the 2nd of August. Daily goals were outlined

to ensure that continuous progress was made on a smaller scale, e.g. to complete subsection 5.1 on the 6th of August. Further project management techniques included monthly group meetings to assess progress and determine areas where further development is needed.

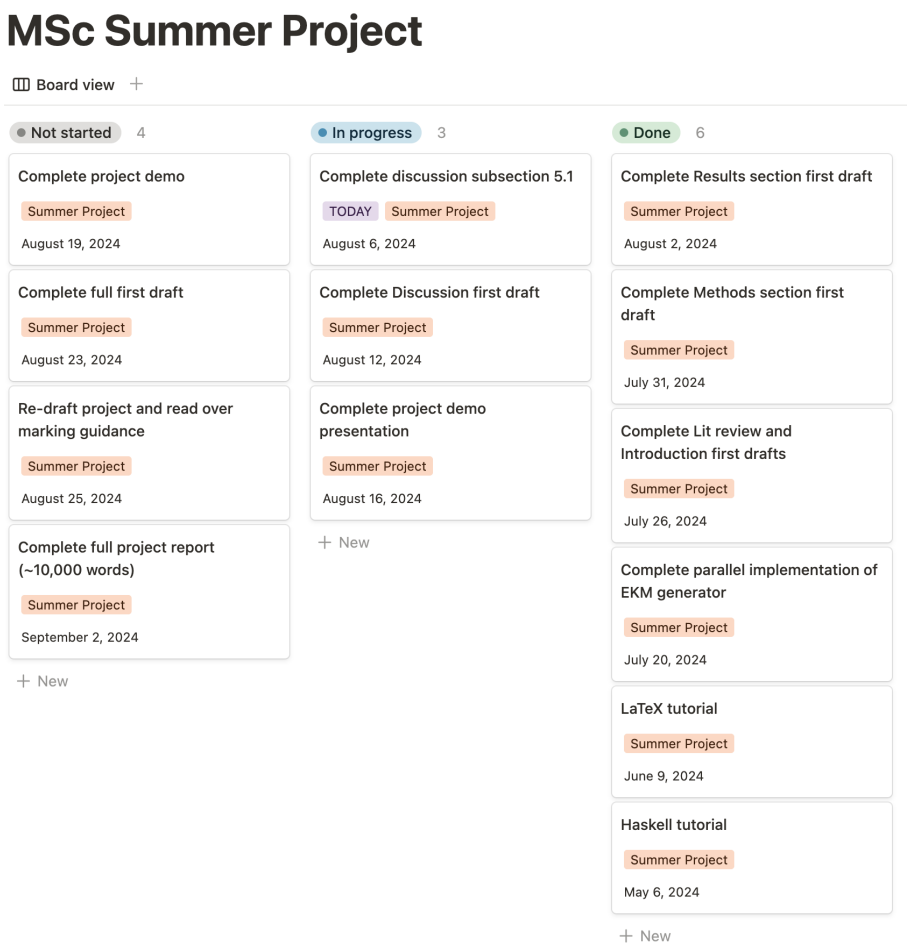


Figure 11: Notion Kanban board used for project management. The three groups are tasks that haven't been started yet, tasks currently in progress and completed tasks. Each task is assigned a due date to give clear deliverables for the project.

3.3 Exact K-medoids Generator

As there is no parallelism taking place in the original EKM generator, there is always just one instance of *gen* required to enumerate all combinations of the

elements of input list N. For example, to generate all combinations of the list $N = [x_1, x_2, x_3, x_4]$ with $K = 2$, the EKM generator works as follows:

$$P1 = gen([x_1, x_2, x_3, x_4], 2) = [[[]], [[x_1], [x_2], [x_3], [x_4]], [[x_1, x_2], [x_1, x_3], [x_1, x_4], [x_2, x_3], [x_2, x_4], [x_3, x_4]]] \tag{14}$$

Where:

- $P1$ is a single instance of *gen*, where the whole input list N is passed in as the first parameter.

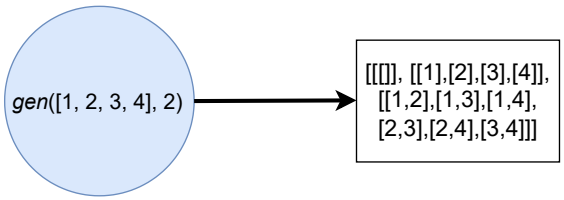


Figure 12: Illustration of the original EKM generator with input list $N = [1, 2, 3, 4]$. Without parallelism, there is just one instance of *gen*, represented by the blue circle. This one function call produces the full result; all combinations of the elements in the input list N.

3.4 Split-Sequential Generator

To eventually parallelise the EKM generator, the input list N must be split up and passed into *gen* separately to instantiate multiple processes. This split-sequential implementation explores the effect of running two instances of *gen* on two partitions of the input list N pre-parallelisation (hence the name 'split-sequential'). Now that there are multiple instances of *gen*, the *merge* function (8) is required to combine their results and enumerate any new combinations between the two partitions of N. With the same input list $N = [x_1, x_2, x_3, x_4]$ with $K = 2$, the split-sequential generator works as follows:

$$\begin{aligned} P1 &= gen([x_1, x_2], 2) = [[[]], [[x_1], [x_2]], [[x_1, x_2]]] \\ P2 &= gen([x_3, x_4], 2) = [[[]], [[x_3], [x_4]], [[x_3, x_4]]] \end{aligned}$$

$$M1 = merge([P1, P2], 2) = [[[]], [[x_1], [x_2], [x_3], [x_4]], [[x_1, x_2], [x_1, x_3], [x_1, x_4], [x_2, x_3], [x_2, x_4], [x_3, x_4]]] \tag{15}$$

Where:

- $P1$, $P2$ are two instances of *gen* executed **sequentially**;
- $M1$ is one instance of *merge* that joins the combinations of $P1$ and $P2$ and enumerates all new combinations to obtain the final result (the set of all combinations of list N).

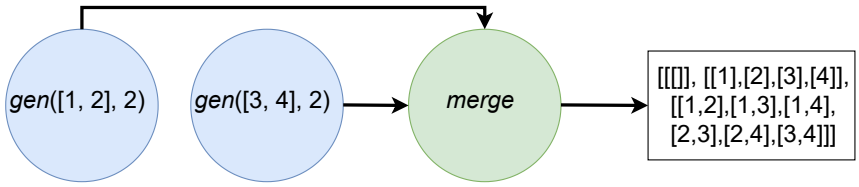


Figure 13: Illustration of the split-sequential generator. Here the first and second halves of N are processed one after another (blue circles). The first blue circle calls *gen* on the sublist [1, 2]. Then the second calls *gen* and passes in [3, 4]. From here their results are merged (green circle), to make sure that all possible combinations are still generated.

3.5 Multiprocessing

The Python Multiprocessing module was used to carry out two parallel multi-core CPU implementations of the EKM generator [57]. The Python Multiprocessing module is a package that supports the creation of multiple processes, that can be programmed to run in parallel. The multithreading module is a common alternative, however, for developing a parallel EKM generator, the Multiprocessing module is preferable for several reasons. By using multiple processes instead of multiple threads, the program is not restricted by the Global Interpreter Lock (GIL), providing utility for CPU-bound algorithms like EKM. EKM is CPU-bound because it is embarrassingly parallel; there is little dependency on input/output operations [1]. The GIL is a common interpreter mechanism that prevents multiple threads from having control over the Python interpreter at a given time. This mutual exclusion protects the critical section of a thread where exclusive access to shared memory is required (see section 1.2.2). Because of EKM's structure, there is no need for critical sections, so multiple processes can execute in parallel without deadlock occurring [14].

The most commonly used feature of the Multiprocessing module is the *Process* class. The *Process* class is what allows the creation and management of many processes. They each can run in parallel and contain their own memory space. This further reduces runtime as individual cache systems enable data storage closer to each processor, reducing memory access time. Parallel implementation is also facilitated by the *Pool* class, which distributes the input data across each core. Within this, the *map* function provides the ability to call the same instruction on each element of the input list in parallel. For example,

if you want to get the square of each number in a list, you can first create a *square* function ($x * *2$), then instantiate a Pool of processes as p. From here you can call p.map(*square*, [1, 2, 3]). This returns the expected output [1, 4, 9], with each squared number calculated in parallel across multiple processes. On the surface, this suggests that you can obtain this output in the time it would take to work out the square of one number sequentially. This is not quite the case, however, due to the overhead of initialising multiple processes and other limitations like context switching and memory usage. For small tasks like the example given, the power and appeal of Multiprocessing is not clear. The most appropriate tasks where dramatic improvements can be seen, are more complex tasks that take longer to execute, like the EKM algorithm.

3.6 Parallel Generator 1

The first multi-core implementation was achieved by instantiating a process pool with two processes to be run in parallel. The first process $P1$ carries out the EKM *gen* function (7) for the first half of N, and the second process $P2$ does the same for the second half of N. Identical to the split-sequential implementation, except now $P1$ and $P2$ are executed in parallel rather than being called sequentially. This enumerates all possible combinations for the first and second halves of N. From here the two results are combined using EKM's *merge* function (8), which also enumerates any new combinations between $P1$ and $P2$, ensuring that the algorithm maintains its global optimality. This step is identical to the corresponding *merge* function taking place in the split-sequential version, as it is yet to be parallelised. Within the *Pool* Multiprocessing class, the *starmap* function is used as an alternative to the *map* function. The *starmap* function works in the same way as *map*, however it allows multiple parameters to be passed into the instruction being called, which is needed for passing in list N and number of medoids K to *gen*. For example, to generate all combinations of the list $N = [x_1, x_2, x_3, x_4]$ with $K = 2$, parallel generator 1 works as follows:

$$\begin{aligned}
&P1 \parallel P2 \\
&P1 = gen([x_1, x_2], 2) = [[[]], [[x_1]], [[x_2]], [[x_1, x_2]]] \\
&P2 = gen([x_3, x_4], 2) = [[[]], [[x_3]], [[x_4]], [[x_3, x_4]]] \\
&M1 = merge([P1, P2], 2) = [[[]], [[x_1]], [x_2], [x_3], [x_4]], [[x_1, x_2], \\
&\qquad\qquad [x_1, x_3], [x_1, x_4], [x_2, x_3], [x_2, x_4], \\
&\qquad\qquad [x_3, x_4]]]
\end{aligned} \tag{16}$$

Where:

- $P1$, $P2$ are two instances of *gen* executed in **parallel**;

- $M1$ is one instance of *merge* that joins the combinations of $P1$ and $P2$, and enumerates all new combinations to obtain the final result (the set of all combinations of list N).

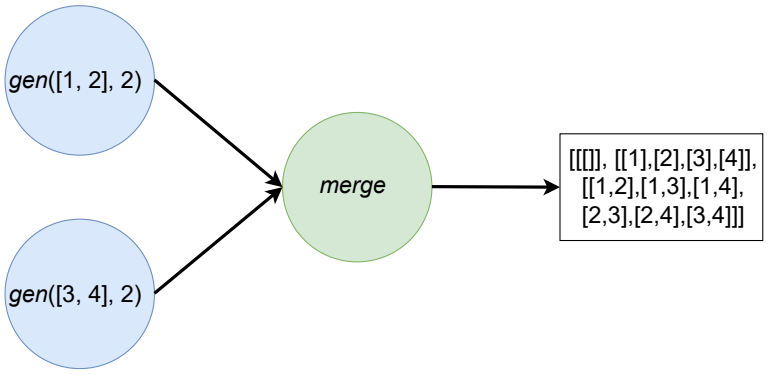


Figure 14: Illustration of parallel generator 1 with list $N = [1, 2, 3, 4]$. The blue circles represent two instances of the *gen* function, enumerating all combinations of each sublist in parallel. The first blue circle calls the *gen* function on the sublist [1, 2] and the second calls *gen* on the sublist [3, 4]. The green circle represents the single *merge* function that joins the two previous results of the blue circles to obtain the final result.

Parallelisation of the *gen* function across input list N increases the overall percentage of parallelisable code. Speedup will be tested for enumerating all combinations in section 4.1. The main limitation of parallel generator 1, is that there is just one merge function taking place, which means that a significant proportion of the generator is still sequential. In addition, as N increases only having two parallel *gen* instances means that large lists ($\frac{N}{2}$) will still be processed sequentially by $P1$ and $P2$ in isolation. This can be improved upon by splitting up the input list N further, allowing for more instances of the *gen* function (blue circles) to take place in parallel. This also means that multiple *merge* operations (green circles) will take place, which can also be parallelised with the Multiprocessing module.

3.7 Parallel Generator 2

To increase the percentage of code being run in parallel, the input list can be split into four parts to be processed in parallel, instead of just two. In addition, the multiple *merge* operations required can also be parallelised. Thus, to generate all combinations of the same list as before, $N = [x_1, x_2, x_3, x_4]$ with $K = 2$, parallel generator 2 works as follows:

$$
\begin{aligned}
&P1 \parallel P2 \parallel P3 \parallel P4 \\
&P1 = gen([x_1], 2) = [[[]], [[x_1]]] \\
&P2 = gen([x_2], 2) = [[[]], [[x_2]]] \\
&P3 = gen([x_3], 2) = [[[]], [[x_3]]] \\
&P4 = gen([x_4], 2) = [[[]], [[x_4]]]
\end{aligned}
$$

$$
\begin{aligned}
&M1 \parallel M2 \\
&M1 = merge([P1, P2], 2) = [[[]], [[x_1], [x_2]], [[x_1, x_2]]] \\
&M2 = merge([P3, P4], 2) = [[[]], [[x_3], [x_4]], [[x_3, x_4]]]
\end{aligned} \tag{17}
$$

$$
\begin{aligned}
M3 = merge([M1, M2], 2) = [&[[]], [[x_1], [x_2], [x_3], [x_4]], \\
&[[x_1, x_2], [x_1, x_3], [x_1, x_4], \\
&[x_2, x_3], [x_2, x_4], [x_3, x_4]]]
\end{aligned}
$$

Where:

- $P1$, $P2$, $P3$, $P4$ are four instances of *gen* executed in parallel;
- $M1$, $M2$ are two instances of *merge* executed in parallel, that join the combinations of [P1, P2], and [P3, P4] respectively, whilst enumerating all new combinations;
- $M3$ is the final instance of *merge* that joins the combinations of $M1$ and $M2$, and enumerates all new combinations.

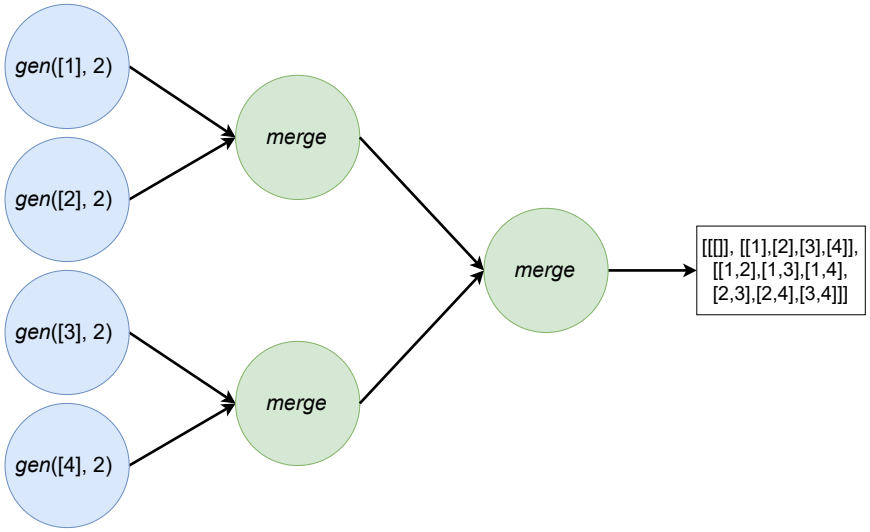


Figure 15: Illustration of parallel generator 2 with the same list $N = [1, 2, 3, 4]$. The blue circles represent the four instances of the *gen* function, enumerating the combinations of each quarter of the input list in parallel. The first two green circles represent two instances of the *merge* function joining the input list's combinations generated by the first two, and the last two blue circles respectively. The first two *merge* instances are executed in parallel before a final *merge* combines their results to arrive at the final set of all combinations of the input list.

By partitioning the input list further and parallelising the merge function, the overall percentage of the generator being executed in parallel has increased. For a list length of just four, like in the example given, increasing the number of parallel processes is unnecessary, but as N increases significant speedup can be achieved, as seen in figure 16. This pattern of subdividing the input list and executing more *gen* and *merge* operations in parallel can be developed further. This technique could, in future, be solved as an optimisation problem. The goal would be to find the optimal partition length of the input list N to be processed sequentially in each parallel *gen* instance, to minimise runtime. This concept will be explored further in section 5.4.

Development of the aforementioned generators and subsequent experiments were carried out with Python and NumPy on the Google Colab Jupyter Notebook platform with a 2-core CPU and the Kaggle Jupyter Notebook platform with a 4-core CPU. In these experiments, the length of the input list N will be denoted as $len(N)$. Apart from figure 10 which was used with the authors' permission, all figures were developed with draw.io and Matplotlib. For further clarity, a link to the GitLab repository is provided in the appendix.

4 Results

4.1 Accelerated Exact K-medoids Generator

For $K = 3$, the empirical runtime of three different generators were compared. Firstly the fully sequential version from the original EKM paper [1]. Then the 'split-sequential' version was tested. This implementation works by running the processes $P1$ and $P2$, one after another (sequentially) before merging. The final implementation is the first parallel EKM generator (parallel generator 1), executed with the Multiprocessing module (see section 3.5). In parallel generator 1 (figure 14), processes $P1$ and $P2$ are executed in parallel before the merge function is called (16).

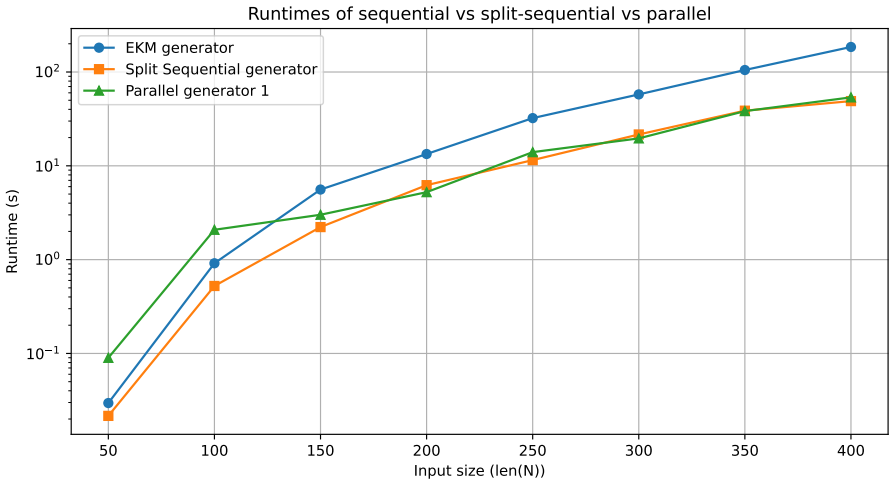


Figure 16: Showing runtimes of the three different EKM implementations ($len(N) = 50 - 400$, $K = 3$). Parallel generator 1 starts as the slowest version due to the initial overhead of instantiating multiple processes mentioned previously. However, it quickly becomes more efficient than the original, sequential EKM generator as $len(N)$ increases. While the parallel generator is the fastest for a number of input sizes, the split sequential generator also performed very well, the fastest in some cases ($len(N) = 250, 400$). These three implementations were all tested on the Google Colab 2-core CPU.

Both the split-sequential and parallel algorithms outperformed the original sequential version as expected, however, the split sequential version was not expected to perform as well as it did. It was expected that both new implementations would provide speedup, however, it was thought that the parallel implementation would outperform the split-sequential implementation as processes $P1$ and $P2$ are executed with the Multiprocessing module.

4.2 2-core vs 4-core: Parallel Generator 1

Next, the performance (runtime) of parallel generator 1 was tested and compared on the Google Colab 2-core CPU and the Kaggle 4-core CPU.

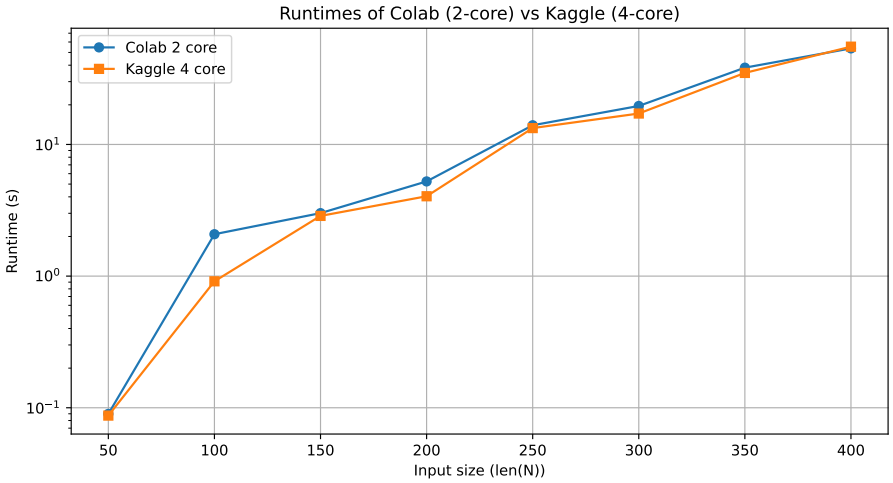


Figure 17: Comparing the runtimes of parallel generator 1 on a Google Colab 2-core CPU against a Kaggle notebooks 4-core CPU ($len(N) = 50 - 400$, $K = 3$).

Both platforms tested provide the necessary resources for parallel speedup of the EKM generator function, however, it was expected that the Kaggle 4-core CPU would perform significantly better than the Google Colab 2-core CPU. For each input size $len(N)$, the Kaggle CPU performed slightly better, except for when $len(N) = 400$; the Google Colab CPU was faster by 1.64s.

4.3 Parallel Generator 1 vs Parallel Generator 2

The runtimes of generator 1 and generator 2 were compared to see the effect of partitioning N into more sublists. Generator 2 splits N into 4 sublists, requiring multiple merge operations to be executed to obtain the full set of combinations (see section 3.7). While this is an increase in the number of stages overall, the number of combinations to be enumerated is the same, with more of generator 2 being processed in parallel:

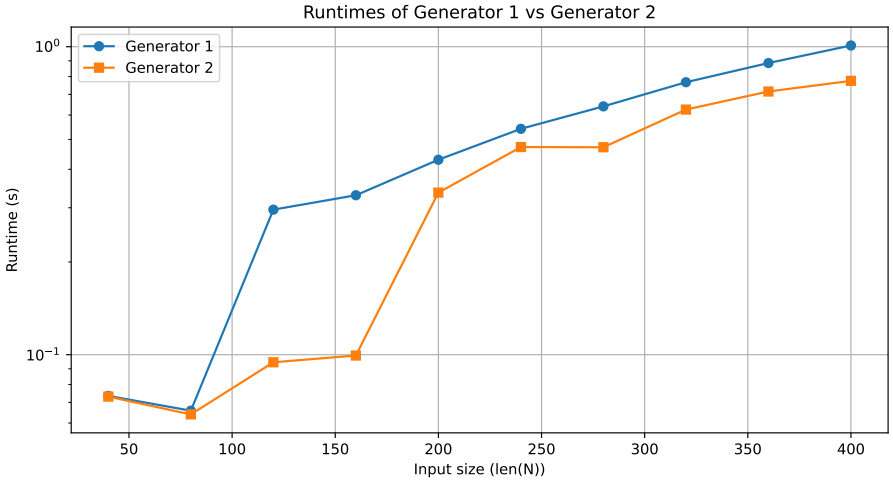


Figure 18: Comparing the runtimes of parallel generator 1 and parallel generator 2. As expected, as $len(N)$ increased generator 2 was consistently faster than generator 1 at enumerating all combinations, because a larger proportion of generator 2 has been parallelised. Both generators were tested on the Google Colab 2-core CPU.

4.4 2-core vs 4-core: Parallel Generator 2

After partitioning the input list N further in parallel generator 2, the runtime was tested again between the Google Colab 2-core CPU and the Kaggle 4-core CPU.

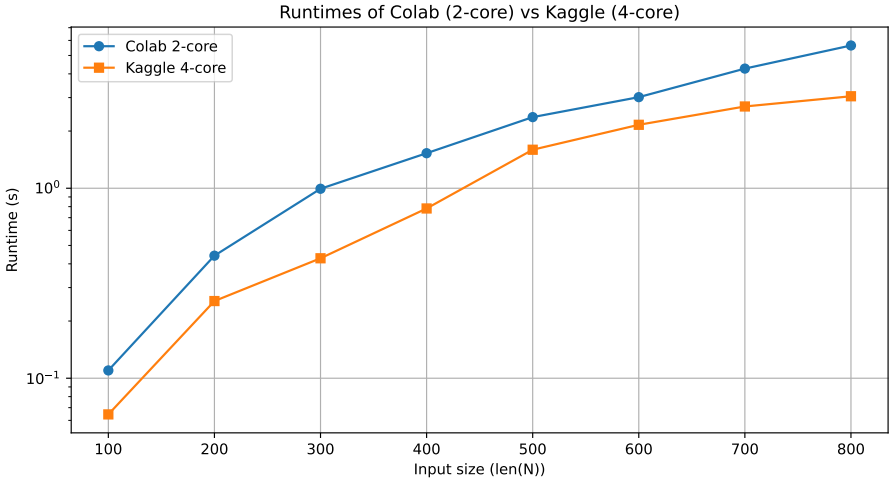


Figure 19: Comparing the runtimes of parallel generator 2 on a Google Colab 2-core CPU and Kaggle 4-core CPU. After initial positive results the runtimes were tested with larger list lengths, $len(N) = 100 - 800$ and the same number of medoids $K = 3$. In contrast to figure 17 and parallel generator 1, significant speedup of parallel generator 2 was achieved by the Kaggle 4-core CPU.

5 Discussion and Evaluation

5.1 Exact *K*-medoids Accelerated by Parallel Multiprocessing

As the predictions and experiments show, the two novel, parallel EKM generators outperform the original sequential generator whilst maintaining optimality and tractability.

Interestingly, the original EKM *gen* function was also significantly accelerated by the 'split-sequential' implementation (figure 13), with it being even faster than parallel generator 1 when $len(N) = 250,400$. This was an unforseen result, however one theoretically aligned explanation, is that multiple function calls allow for implicit parallelism to take place without it being explicitly encoded like in the parallel multiprocessing version. One example is instruction-level parallelism (see section 1.2.4), where modern processors implicitly carry out multiple independent operations in parallel [4]. Because $P1$ and $P2$ (15) are not dependent on each other's results, the CPU is able to execute instructions from each process in parallel, even when called sequentially. Furthermore, in multi-core CPUs, different function calls e.g. $P1$ and $P2$, may automatically be assigned to different cores. At the hardware level, different parts of the CPU such as multiple arithmetic logic units (ALUs) can work on different instructions at the same time [58].

These new parallel generators are important for reducing the runtime of EKM, but its wider effects are also significant. For example, the 2016 study by Han et al [21] mentioned in section 1.2.4, uses K-means clustering to determine weight sharing for each layer of a trained neural network. This was part of a larger 3-stage method for compressing large models while maintaining accuracy. Specifically, they partitioned N weights $W = \{w_1, w_2, ..., w_N\}$ into K clusters $C = \{c_1, c_2, ..., c_K\}, N > K$. This is to minimize $E_{K-clustering}$ (5), for which they used the following sum of squares distance function:

$$\underset{C}{arg\,min} \sum_{k=1}^{K} \sum_{w \in c_k} |w - c_k|^2 \tag{18}$$

This method of determining weight sharing via clustering allows for more informed decisions on which weights should share the same value, maintaining a more accurate representation of the original network. This is because clustering can take place after a network has been fully trained, and so the shared weights are an approximation of the original network. Previous methods have determined weight sharing via a hash function, which takes place before any training has begun. This is an important distinction, which is key for techniques such as transfer learning; the use of large pretrained models is becoming increasingly popular with more companies turning to open-source development [59].

The authors identify that centroid initialisation is instrumental in the quality of the weight sharing clusters produced. This is because K-means is an approximate method, and the initial choice of where to place centroids determines how the algorithm will converge. Thus, the work by Han et al [21] is a prime example of the broader impact EKM can have. EKM's optimality and tractability ensures that an exact solution can always be found for weight sharing, in polynomial time (with respect to K). This will significantly improve the authors' deep neural network (DNN) compression pipeline. In addition, this optimisation with EKM will boost their impact on DNNs in mobile applications where size and bandwidth are limited. This is just one area of AI research that EKM can have a significant effect. An important point to be highlighted, however, is that EKM is efficacious when the number of medoids K (number of shared weights) is low. If the desired number of shared weights is high for the network, then the EKM algorithm will become intractable for Han et al's deep compression pipeline [21]. This is because EKM's complexity is $O(N^{K+1})$.

While CPUs can be slower than GPUs for machine learning based tasks, many individuals or even whole organisations struggle to gain access to high-end GPUs, with cost and availability presenting difficulties. Until recently, the average wait time for NVIDIA's high-end H100 GPUs was 8-11 months [60]. In a rapidly developing field like machine learning, many architectural developments may have been made in this wait time alone. CPUs are often considered to be more suitable for particular tasks involving smaller datasets or models, which may not benefit from expensive, large-scale GPU architectures. While this is true, a previously highlighted study by Wang et al [45] also found that CPUs were more effective for training large, FC neural networks.

5.2 Number of Cores vs Number of Processes

The EKM algorithm is largely parallelisable, and so it was a surprising result that the Kaggle 4-core architecture did not significantly outperform the Google Colab 2-core architecture when tested with parallel generator 1 (figure 17). The main reason for this becomes clear, however, when looking at the structure of parallel generator 1 (figure 14). While parallel processing has been implemented effectively, just two processes are running in parallel ($P1$ and $P2$). Each of these processes can run on a separate CPU core, making full use of a 2-core system, however, they are unable to make use of additional cores. In general, to fully utilize a system with multiple cores, there should be at least as many processes as there are cores.

The limitations of processor scaling are also underpinned by Amdahl's law (figure 3), where a plateau in performance is largely dependent on the percentage of parallelisable code. While parallel generator 1's performance is constrained by this law and its own structure, it was expected that processor scaling would

initially provide an improvement in performance, an effect that can be seen early on in Amdahl's law, especially due to EKM's inherently parallel structure. However, this was not seen for parallel generator 1 when scaling the number of processors n from just 2 to 4. Another reason for this is that a single *merge* function is used in parallel generator 1 and is not implemented with the Multiprocessing module like in parallel generator 2. As a result the percentage of parallelisable code is limited. Highlighted in figure 3, the percentage of parallelisable code is the biggest constraint for predicted speedup achieved when increasing the number of processors available; for example having just 50% parallelisable code caps the maximum theoretical speedup to $2\times$ even as the number of processors $n \to \infty$.

Despite the shortcomings of processor scaling when applied to parallel generator 1, parallel generator 2 benefited from significant speedup (figure 19). As eluded to, this is largely due to ensuring that there are at least as many processes running in parallel as there are cores. As parallel generator 2 partitions the input list N into 4 parallel processes ($P1 - P4$), the 4-core CPU can now be fully utilised. Furthermore, having multiple merge functions in parallel ($M1$ and $M2$) means that the overall percentage of parallelisable code has been increased. These architectural improvements are also highlighted in figure 18, where performance was compared to parallel generator 1. Another limitation seen with parallel generator 1, is that when N is large, the cache memory assigned to each process can become full. In this scenario, full parallelism visualised in figure 14 cannot be achieved. In such cases, parallel generator 1's structure will be closer to that of the split-sequential generator. This is another key factor for the utility of parallel generator 2, and future work where N can be partitioned further across more parallel processes, allowing for better management of each process' cache memory.

A notable limitation of this study is the difference in platform infrastructure. Google Colab and Kaggle run times depend on overall service demands and resource allocation, making it difficult to compare their performance with high accuracy and confidence. This is a common drawback of cloud-based services, and direct examples of differentiable runtimes can be seen between my experiments ran at different times across the project. For example, the runtime of parallel generator 1 was calculated as part of figures 16, 17 and figure 18, for inputs $len(N) = 50 - 400$ and $K = 3$. The experiments that produced figures 16 and 17 were carried out at a similar stage in time, and show similar runtimes. In more detail, for $len(N) = 400$, figure 16 and 17 report a runtime of just over 50 seconds. In contrast, in what was initially thought to be an error, figure 18 shows a runtime of just 1.01 seconds for an input $len(N) = 400$. This was recorded a few weeks later than the previous results, and upon running the first experiment again it was found that it returned similar, much faster runtimes for parallel generator 1. While this volatility may make comparisons between experiments difficult, one positive from this finding, is that analysis within individual experiments can be conducted accurately. This is because tests within

each experiment were conducted concurrently.

5.3 Speedup Prediction Models

A line of best fit was calculated for each implementation to be able to compare the performance of parallel generators 1 and 2 with the original EKM generator, for any input list length. This was achieved with the linear regression calculator on GraphPad [61]. This comparison allows speedup to be predicted for each parallel generator, given a particular length of input list ($len(N)$). As mentioned previously, overall cloud performance was slower when runtimes were recorded for figure 16, thus new runtimes for each generator were recorded at this stage to mitigate any bias. The three models fitted to the runtimes of each generator are as follows:

$$\begin{aligned} \text{Exact } K\text{-medoids} &: y = 0.4474x - 51.59 \\ \text{Parallel Generator 1} &: y = 0.1842x - 18.69 \\ \text{Parallel Generator 2} &: y = 0.1704x - 18.08 \end{aligned} \tag{19}$$

Where:

- y is the runtime in seconds;
- x is the length of input list N.

From here, predicted speedup can be calculated by computing the difference in runtime between two generators, and then representing this difference as a percentage of the original EKM generator.

For example, with an input list length of 10,000, predicted runtimes are 4422.41s, 1823.31s and 1688.92s for the EKM generator, parallel generator 1 and parallel generator 2 respectively. In terms of the EKM generator, parallel generator 1 is 59% faster, and parallel generator 2 is 62% faster, for an input list of length 10,000. To convert these percentages to $\times$ speedup, the following equation is used:

$$\times \text{ speedup} = \frac{100}{100 - \%\text{ speedup}} \tag{20}$$

This is the final computation required which calculates a 2.44$\times$ speedup provided by parallel generator 1 and a 2.63$\times$ speedup provided by parallel generator 2. One key factor to keep in mind is that runtimes were collected from experiments where the number of medoids K=3, so for users that want to predict speedup for any number of medoids, similar methodology could be replicated

with more runtime data for other values of K. Another limitation of this technique is that mathematically predicting empirical runtime is difficult because there are many functional interactions between the software and hardware components of the system which are unable to be accounted for. One example of this is the volatility of runtimes on cloud-based services highlighted previously. Upon repetition, more advanced models could be fit to each set of runtimes. For example, higher polynomials of x may provide more accuracy when representing each generator's performance, rather than a linear model. More extensive data collection would also improve speedup prediction model accuracy.

5.4 Future Work

5.4.1 Optimal Parallel Generator

Performance improvement was seen going from parallel generator 1 to parallel generator 2 (figure 16), where the number of partitions of input list N doubled, and multiple merge functions were parallelised. One way of developing this approach further would be to investigate and determine the optimal list length to be processed sequentially, depending on the length of input list N. This will provide insight into the best way to partition input list N for parallel processing. Another downstream variable to be considered would be the number of *merge* functions taking place in parallel, and how this also impacts runtime. For example, if the optimal list length to be processed sequentially was found to be 10, then for any application of the generator where the length of input list $len(N) \leq 10$, there can be one *gen* instance provided with the whole list, like the original EKM generator. Lists of length > 10 can be split into partitions of size 10 with any remainder taken into account as an additional instance of *gen*. Further investigation of this optimal value may lead to greater speedup than previously seen when running the algorithm on the Kaggle 4-core CPU, as the limitations imposed by sequential code will be reduced.

Following this, the number of cores available to the parallel generator could also be investigated, with the same goal of minimising runtimes. This may also have implications for what the optimal length of each partition of N should be.

5.4.2 Derivation of Parallel Exact *K*-medoids

Because the *eval* and *sel* functions are fused into the *gen* function (see section 2.1.1), speedup achieved through generator parallelism can, in future work, be directly applied to derive a parallel EKM algorithm as a whole. Using either of the parallel generators proposed, the same equational reasoning steps used for EKM can be applied to ensure retention of optimality and tractability. Because *gen* is still recursive, *eval* can still be calculated cumulatively as a sum of loss terms as in the original EKM theory [1]. This means that *eval* can be fused with *gen* and the loss calculated at the same time combinations are enumerated by the parallel generator:

$$\begin{aligned} s^* &= sel_E(eval_E(P[gen_{combs}(n,k)])) \\ &= sel_E(P[evalgen_{E,combs}(n,k)]) \end{aligned} \tag{21}$$

Where:

- P represents the algorithm being processed in parallel with the Multiprocessing module.

Because the objective value is now calculated alongside each combination generation, the best combination (with respect to the objective value, calculated by *eval*) can be stored at each stage. This allows for the fusion of *sel* into $P[evalgen_{E,combs}]$. This is the final fusion needed to derive the parallel EKM algorithm:

$$\begin{aligned} s^* &= sel_E(P[evalgen_{E,combs}(n,k)]) \\ &= P[selevalgen_{E,combs}(n,k)] \\ &= P[EKM(n,k)] \end{aligned} \tag{22}$$

6 Conclusion

Beyond K-medoids clustering, there are a number of other algorithms that would experience significant benefits from parallel computing. For example, Xi He et al [62] recently published another exact and tractable algorithm, this time solving the 0-1 loss linear classification problem. They state in their future work section that for instances where the number of dataset examples N or dimensions D are large, practical implementation of their algorithm can be achieved by creating a parallel version that can run across massively parallel GPUs.

In summary, the results presented demonstrate the significant effect of multi-processing on reducing the EKM generator's runtime. The impact of multi-core CPUs and core scaling has also been elucidated, with the relative number of parallel processes proving to be a key variable. This technique of splitting up the input list N into more partitions should be investigated further, along with core scaling, with the potential of transitioning to multiple, multi-core CPU systems. Restructuring the EKM algorithm would also allow for investigation of parallel GPU execution, providing speedup for industry applications where this architecture is available. Overall, continued improvements in EKM generator runtime will provide significant benefits for critical use cases of K-clustering, where fast arrival at the globally optimal solution is of utmost importance.

References

[1] He X, Little MA. EKM: An exact, polynomial-time algorithm for the K-medoids problem; 2024. Available from: https://arxiv.org/abs/2405.12237.

[2] Introduction to Parallel Computing Tutorial; 2024. Available from: https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial.

[3] Herz D. A Century of Moore's Law; 2023. Available from: https://www.semianalysis.com/p/a-century-of-moores-law.

[4] Hennessy JL, Patterson DA. Computer architecture : a quantitative approach. Cambridge, Ma: Morgan Kaufmann; 2019.

[5] GPU boost technology geforce;. Available from: https://www.nvidia.com/en-gb/geforce/technologies/gpu-boost/technology/.

[6] Flynn LJ. Intel Halts Development Of 2 New Microprocessors. The New York Times. 2004 May. Available from: https://www.nytimes.com/2004/05/08/business/intel-halts-development-of-2-new-microprocessors.html.

[7] Le Sueur E, Heiser G. Dynamic voltage and frequency scaling: The laws of diminishing returns. In: Proceedings of the 2010 international conference on Power aware computing and systems; 2010. p. 1-8.

[8] Saulsbury A, Pong F, Nowatzyk A. Missing the memory wall: the case for processor/memory integration. In: Proceedings of the 23rd Annual International Symposium on Computer Architecture. ISCA '96. New York, NY, USA: Association for Computing Machinery; 1996. p. 90–101. Available from: https://doi.org/10.1145/232973.232984.

[9] Efnusheva D, Cholakoska A, Tentov A. A survey of different approaches for overcoming the processor-memory bottleneck. International Journal of Computer Science and Information Technology. 2017;9(2):151-63.

[10] Rauber T. Parallel programming : for multicore and cluster systems. Springer; 2014.

[11] Amdahl GM. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. IEEE Solid-State Circuits Newsletter. 2007;12(3):19–20.

[12] Gustafson JL. Reevaluating Amdahl's law. Commun ACM. 1988 may;31(5):532–533. Available from: https://doi.org/10.1145/42411.42415.

[13] Bernstein AJ. Analysis of Programs for Parallel Processing. IEEE Transactions on Electronic Computers. 1966 Oct;EC-15(5):757–763. Available from: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4038883.

[14] Lutkevich B. What is a race condition?; 2021. Available from: https://www.techtarget.com/searchstorage/definition/race-condition.

[15] What is Parallel Computing? Definition and FAQs;. Available from: https://www.heavy.ai/technical-glossary/parallel-computing.

[16] Kumar Yadav M. Performance Analysis of Serial Computing Vs. Parallel Computing in MPI Environment. International Journal of Advanced Research in Science, Communication and Technology (IJARSCT); 2022.

[17] Parallel Slowdown. Semantic Scholar;. Available from: https://www.semanticscholar.org/topic/Parallel-slowdown/2161443.

[18] Bellman R. The theory of dynamic programming. Bulletin of the American Mathematical Society. 1954;60(6):503-15.

[19] Flynn MJ. Some Computer Organizations and Their Effectiveness. IEEE Transactions on Computers. 1972 Sep;C-21(9):948–960. Available from: https://www.cs.utah.edu/~hari/teaching/paralg/Flynn72.pdf.

[20] Yang H, Duan L, Chen Y, Li H. BSQ: Exploring Bit-Level Sparsity for Mixed-Precision Neural Network Quantization; 2021. Available from: https://arxiv.org/abs/2102.10462.

[21] Han S, Mao H, Dally WJ. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding; 2016. Available from: https://arxiv.org/abs/1510.00149.

[22] Krizhevsky A, Sutskever I, Hinton GE. ImageNet Classification with Deep Convolutional Neural Networks. In: Pereira F, Burges CJ, Bottou L, Weinberger KQ, editors. Advances in Neural Information Processing Systems. vol. 25. Curran Associates, Inc.; 2012. Available from: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[23] Ulfsnes R. Design of a Snoop Filter for Snoop Based Cache Coherency Protocols; 2013. Available from: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2370790.

[24] Ghosh S. Distributed systems : an algorithmic approach. Boca Raton: Crc Press, Taylor & Franciss Group; 2015.

[25] What is a computing cluster?;. Available from: https://www.supermicro.com/en/glossary/computing-cluster.

[26] What is grid computing? grid computing explained;. Available from: https://aws.amazon.com/what-is/grid-computing/.

[27] NVIDIA. CUDA Zone; 2019. Available from: https://developer.nvidia.com/cuda-zone.

[28] Plank JS, Beck M, Kingsley G, Li K. Libckpt: Transparent Checkpointing under UNIX. In: USENIX 1995 Technical Conference (USENIX 1995 Technical Conference). New Orleans, LA: USENIX Association; 1995. Available from: https://www.usenix.org/conference/usenix-1995-technical-conference/libckpt-transparent-checkpointing-under-unix.

[29] Moore GE. Cramming More Components onto Integrated Circuits. Proceedings of the IEEE. 1998 Jan;86(1):82–85.

[30] Roser M, Ritchie H, Mathieu E. What is Moore's Law? Our World in Data. 2023. Https://ourworldindata.org/moores-law.

[31] Dennard RH, Gaensslen FH, Yu HN, Rideout VL, Bassous E, LeBlanc AR. Design of ion-implanted MOSFET's with very small physical dimensions. IEEE Journal of Solid-State Circuits. 1974 Oct;9(5):256–268.

[32] Stulen RH, Sweeney DW. Extreme ultraviolet lithography. IEEE Journal of Quantum Electronics. 1999 May;35(5):694–699.

[33] Moore's Second Law;. Available from: https://dbpedia.org/page/Moore's_second_law.

[34] Intel research fuels moore's law. Intel;. Available from: https://www.intel.com/content/www/us/en/newsroom/news/moores-law-paves-way-trillion-transistors-2030.html.

[35] What is an AI accelerator?. Synopsis;. Available from: https://www.synopsys.com/ai/what-is-an-ai-accelerator.html.

[36] Rose J, El Gamal A, Sangiovanni-Vincentelli A. Architecture of field-programmable gate arrays. Proceedings of the IEEE. 1993 Jul;81(7):1013–1029.

[37] AMD. What is an FPGA? Field Programmable Gate Array; 2019. Available from: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html.

[38] Deep learning processors: which one is right for you?. ECS; 2022. Available from: https://www.equuscs.com/deep-learning-processors-which-one-is-right-for-you/.

[39] Introduction to cloud tpu. Google Cloud;. Available from: https://cloud.google.com/tpu/docs/intro-to-tpu.

[40] Jouppi N. Google supercharges machine learning tasks with TPU custom chip; 2016. Available from: https://cloud.google.com/blog/products/ai-machine-learning/google-supercharges-machine-learning-tasks-with-custom-chip.

[41] Jouppi N, Young C, Patil N, Patterson D. Motivation for and Evaluation of the First Tensor Processing Unit. IEEE Micro. 2018 May;38(3):10–19.

[42] Jumper J, Evans R, Pritzel A, Green T, Figurnov M, Ronneberger O, et al. Highly accurate protein structure prediction with alphafold. Nature. 2021 Jul;596(7873):583–589. Available from: https://www.nature.com/articles/s41586-021-03819-2.

[43] Google health AI;. Available from: https://ai.google/discover/healthai/.

[44] Dally WJ, Keckler SW, Kirk DB. Evolution of the Graphics Processing Unit (GPU). IEEE Micro. 2021 Nov;41(6):42–51. Available from: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9623445.

[45] Wang Y, Wei GY, Brooks DJ. Benchmarking TPU, GPU, and CPU Platforms for Deep Learning. arXivorg. 2019 Jul.

[46] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, et al.. Attention Is All You Need; 2023. Available from: https://arxiv.org/abs/1706.03762.

[47] Lloyd S. Least squares quantization in PCM. IEEE Transactions on Information Theory. 1982;28(2):129-37.

[48] Jain AK, Dubes RC. Algorithms for clustering data. Prentice-Hall, Inc.; 1988.

[49] Kaufman L, Rousseeuw P. Partitioning Around Medoids (Program PAM). Finding Groups in Data. 1990 Mar:68–125.

[50] Land AH, Doig AG. An Automatic Method of Solving Discrete Programming Problems. Econometrica. 1960 Jul;28(3):497.

[51] Meertens L. Towards programming as a mathematical activity. Proceedings of the CWI Symposium on Mathematics and Computer Science. 1986;1:289-334.

[52] Schubert E, Rousseeuw PJ. Fast and eager k-medoids clustering: O(k) runtime improvement of the PAM, CLARA, and CLARANS algorithms. Information Systems. 2021 Nov;101:101804.

[53] Ren J, Hua K, Cao Y. Global optimal k-medoids clustering of one million samples. Advances in Neural Information Processing Systems. 2022;35:982-94.

[54] Makhorin A. GLPK (gnu linear programming kit); 2008. Available from: https://www.gnu.org/software/glpk/glpk.html.

[55] Sethi A, Kushwah H. Multicore processor technology-advantages and challenges. International Journal of Research in Engineering and Technology. 2015;4(09):87-9.

[56] Walls C. Multicore System - an overview — ScienceDirect Topics; 2012. Available from: https://www.sciencedirect.com/topics/computer-science/multicore-system.

[57] Python. multiprocessing — Process-based parallelism — Python 3.8.3rc1 documentation; 2024. Available from: https://docs.python.org/3/library/multiprocessing.html.

[58] Difference between implicit parallelism and explicit parallelism in parallel computing. GeeksforGeeks; 2023. Available from: https://www.geeksforgeeks.org/.

[59] Zuckerberg M. Open Source AI Is the Path Forward; 2024. Available from: https://about.fb.com/news/2024/07/open-source-ai-is-the-path-forward/.

[60] Klotz A. Nvidia's H100 AI GPU shortages ease; 2024. Available from: https://www.tomshardware.com/pc-components/gpus/.

[61] GraphPad. GraphPad QuickCalcs: Linear Regression Calculator; 2018. Available from: https://www.graphpad.com/quickcalcs/linear1/.

[62] He X, Rahman WU, Little MA. An efficient, provably exact, practical algorithm for the 0-1 loss linear classification problem; 2023. Available from: https://arxiv.org/abs/2306.12344.

Appendix

GitLab repository: `https://git.cs.bham.ac.uk/projects-2023-24/bxm043`

The main branch of the repository includes two Jupyter Notebook files called 'Generator_parallelV2.ipynb', and 'Generator_results.ipynb'. The contents of 'Generator_parallelV2.ipynb' includes the original EKM auxiliary functions, the original EKM generator (both used with permission from the authors) as well as the new generators developed and their various experiments. To run the experiments, work through and run each code cell in chronological order in the notebook. To run the experiments with different input list lengths or number of medoids, change the values assigned to variables 'list_length' and 'K'. When experimenting with larger list lengths, it is recommended to comment out print statements of configurations enumerated by each generator.

'Generator_results.ipynb' includes the data collected (runtimes) for each experiment. It also includes the Matplotlib code used to develop the figures in the results section.

As well as the main branch, more Jupyter Notebook files have been added that were used to create supporting figures with Matplotlib. These can be found in the 'additional_figures' branch of the repository.