Adaptive Manta Ray Foraging Optimizer for Determining Optimal Thread Count in Multi-threaded Applications

SACHIN H MALAVE[1]
SUBHASH K SHINDE[1]

Mumbai University
Lokmanya Tilak College of Engineering
Koparkhairane, New Mumbai 400709
[1]shmalave@apsit.edu.in, skshinde@rediffmail.com

Abstract. In high-performance computing, selecting the appropriate thread count has a significant impact on execution time and energy consumption. On multi-core processor systems, it is widely believed that for maximal speedup, the total number of threads should match the number of cores. Thread migration rate, cache miss rate, thread synchronisation, and context switching rate are all impacted by changes in thread count at the hardware and OS levels. As a result, it is extremely difficult to analyse these factors for relatively complex multi-threaded programs and determine the optimal number of threads. The method put forward in this study is an enhancement of the conventional Manta-Ray Foraging Optimization, a bio-inspired approach that has been applied to a number of numerical engineering problems. The proposed approach makes use of three foraging steps: chain, cyclone, and somersault. Using the well-known benchmark suite PARSEC (The Princeton Application Repository for Shared-Memory Computers), the suggested work is simulated on an NVIDIA-DGX Intel Xeon-E5 2698-v4 processor. The findings demonstrate that the new modified AMRFO-based prediction model can choose the appropriate number of threads with fairly minimal overheads when compared to the current method.

Keywords: Multi-threading, Manta-Rays, Optimization, Nature-inspired, Parallel programs

(Received May 12th, 2022 / Accepted Dec 11th, 2022)

1 Introduction

Since engineering and scientific applications are now more data-intensive, it has become necessary to build new high-performance computer systems as well as approaches that are able to efficiently execute a large number of tasks using the processors that are already available.The term "multi-threading" refers to the practice of employing parallel programming techniques on a multi-core processor that uses shared memory. It is the ability of the Central Processing Unit (CPU) and the operating system to carry out several threads at the same time[35]. The basic objective of multi-threading is to enable the concurrent execution of two or more sections of code in order to increase the amount of work that can be done by a single CPU. In the context of this discussion, every component of a program is referred to as a thread. Programming nowadays makes extensive use of a technique known as multi-threading, which is a relatively recent development in the field of information technology.Programmers employ many threads for a variety of purposes, including designing responsive servers that can connect with several clients, doing complex calculations simultaneously on a multiprocessor for improved performance, and constructing sophisticated user interfaces[29]. It is essential to have High-Performance Computing (HPC) development tools in order to simplify the process of programming on HPC computers. Programmers have access to a wide variety of tools that are designed to assist them in the creation of effective parallel programs [5]. Because of this, multiprocessors have become standard in every variety of computer system, ranging from desktop computers to supercomputers, throughout the course of the last few decades.

The OpenMP programming language has recently been updated to provide a more extensive collection of directives, which cover a wider range of parallelisation choices than merely shared memory. When we examine the future of OpenMP, we envision a further extension of support for a range of parallelisation techniques, as well as the introduction of support for performance monitoring and debugging tools. This is something that we are looking forward to seeing[7]. HPC has experienced a huge increase in its computing capability as a direct result of the relatively recent development of general-purpose graphics processing units (GPGPU) as a low-cost processor architecture. Nvidia has recently introduced CUDA for programming GPGPUs, which will significantly reduce the amount of pressure placed on programs. C/C++ application programming interfaces (APIs), tools, and a hardware abstraction technique are all part of CUDA, which is designed to let programmers run parallel programs on GPGPUs[10]. When it comes to distributed computing, there are a lot of different strategies and methods that are developed to help increase the performance of the systems. It is vital to conduct an analysis of the environment of network-based distributed computing in order to determine the performance of the systems[12].

A thread is also referred to be a lightweight subprocess that can collaborate with other threads of the concurrent operation[39]. Since parallelism is executed on a variety of levels, pinpointing its precise nature can be challenging. In addition, the speed of execution is increased by doing multiple computations at the same time[18]. In addition, the process of parallelising code might result in other problems within the program such as deadlocks, race situations, bugs, and so on. A few different modes of communication are required for separate works[16][1]. This communication creates an additional burden, which may result in a decrease of performance. When a great number of threads make simultaneous use of the CPU cache, it can lead to overflow and a high cache miss rate. [43].

The major part of the bandwidth utilisation, thread migration rate, number of threads, cache miss rate, and context transition rate are dominated by factors related to the design of the hardware and the operating system. Because changing the number of threads affects how other parameters behave, it may be said that the number of threads is the most important element. The higher the total number of threads, the higher the cache miss rate will be since more threads will be able to utilise the shared cache. Additionally, a greater number of threads are able to interact with the shared bandwidth; as a result, there are more transmission delays.

There are some programs that require a significant amount of work, and increasing the number of threads in those programs can help achieve good performance. There are some applications that are designed to be extremely memory intensive, and because of a shortage of storage bandwidth and shared storage capacity, creating additional threads does not improve the performance of these programs. Certain computer applications have stringent communication requirements, which need constant data exchange between different threads. In addition, enabling more threads results in a significant increase in the overhead required for lock synchronisation, which in turn significantly hinders performance. Therefore, in order to achieve good performance, the approach that has been suggested obtains the thread count dynamically. This is done based on the characteristics of the application.

Even if parallel processing on multi-core processors is a more developed technology, it is still important to determine the exact level of concurrency that should be utilised for a certain work in order to enhance the average CPU utilisation of multi-core processors for a particular amount of time[21] [26]. This can be accomplished by determining the optimal level of parallelism that should be utilised for the task in question. It is impossible to accurately forecast the performance of a program that contains a large number of software threads without first carrying out the tasks with many threads in their entirety; this is an essential component that enables efficient CPU use[20] [3]. The ever-increasing complexity of the system architecture makes it an extremely challenging endeavor to build programs and make accurate projections on their expected performance. Estimating the thread count using traditional logical performance methods is the most accurate way to do so because they are based on the characteristics of both the hardware and the software[25]. However, they are not perfect by any means and require at least some implementation details of the target processor in order to gather hardware-specific indications or implementation statistical data such as memory traces and instruction counts, both of which have the potential to influence the forecasting time[17]. Researchers have been able to solve problems with credit card authorization systems with the assistance of parallel programs that make use of many threads, which has also allowed them to parallelise existing serial applications in order to improve their performance[13].

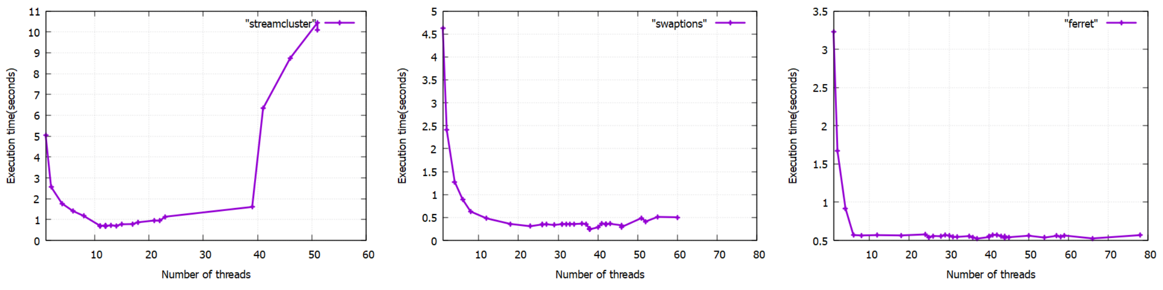

Figure 1: Execution time taken by the PARSEC benchmarks streamcluster(left), swaptions(middle) and ferret (right) for sample input data

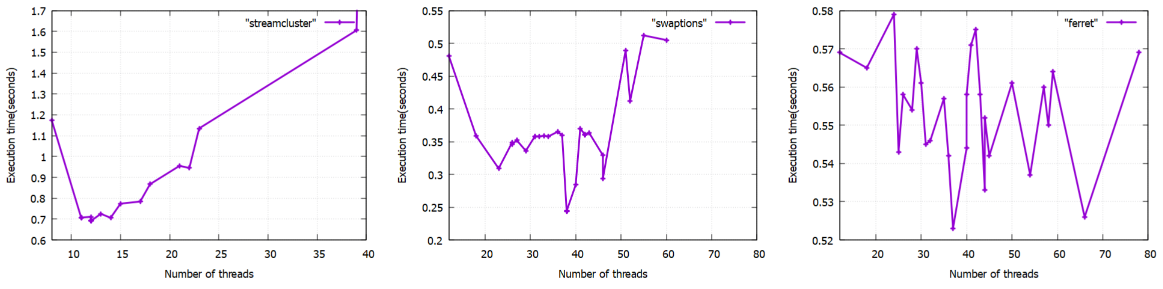

Figure 2: Zoomed in area of Figure 1 showing regions where benchmarks were unable to scale with the number of threads

To compute the performance of applications running on specific hardware and with varying degrees of parallelism, a variety of different methodologies have been devised[36]. In order to achieve this goal, characteristics of applications that correlate to a single thread are mined in order to predict speedups for a variety of different numbers of threads[27]. These features are obtained after the program of interest has been run on the target machine for a single thread for a certain amount of time. The varied multi-threaded performance of the application is evaluated on the desired hardware, and the speedups are calculated based on the attributes that are derived from it[11]. Following the completion of a single-threaded version of the task at hand, this enables multi-threaded application programs to determine the optimum number of threads that should be utilised in order to complete the task. This is not supported for all programs that are envisioned to be implemented only once or twice over the course of their lifetime since the application needs to be implemented just once on a single thread in order to mine features for determining the best possible number of threads[19]. Therefore, the most difficult challenge for applications that are intended to be executed a great number of times is to determine the appropriate number of threads to use.

In order to find a solution to such an optimization problems, bio-inspired algorithms may be utilised. The way that live things function served as a model for the development of these algorithms. These methods can also be used to the field of wireless sensor networks in order to make more effective use of the available energy[24]. The Partical Swarm algorithm is very famous for solving the optimization problem. A study by Deep et al.[8] shows the use this bio-inspired algorithm in the field of Lennard-Jones Problem.

2 Motivation & Problem Statement

Since computers are now used in every facet of our life, the tasks that may be carried out by mainframes and PCs are becoming extraordinarily complicated. Additionally, there has been a considerable growth in the volume of data that can be successfully managed by today's computers [31, 32, 33, 23, 40, 38, 15, 30, 6, 22, 34]. Personal computers of today often include anything from two to eight cores, which enables a large number of tasks, or threads, to be carried out all at once. Additionally, if an application has an excessive number of threads, this can lead to contention risks with other application threads, such as poor cache locality, thread migrations, and stealing CPU time from other threads. On the other hand, if the application has an inadequate number of threads, the resources will be wasted. Also because number of threads that are formed by a program at the time of execution is always increasing, managing threads is becoming an increasingly difficult task. Synchronisation, Task Granularity, Load Balancing, Data sharing, Resource sharing, Data Locality, and Input or Output are the seven categories that are used to classify the parallel programming issues that arise in multi-core systems. It is necessary for the operating system to solve all of these issues in order to ensure that tasks are carried out correctly. In the context of thread management, these issues are sometimes referred to as overheads. When the number of threads increases, the overhead costs do as well, and it's possible that the costs of overhead will end up being higher than the profits. Therefore, determining the ideal number of threads for the situation at hand is a difficult task to undertake.

The figure 1 displays the amount of time that streamcluster, swaptions, and ferrets required to complete their respective tasks when given sample input data on a computer system that contained 40 cores of Xeon processors. The X-axis indicates the number of threads that are now being utilised, and the Y-axis indicates the amount of time required by the program to finish executing itself at the thread count that has been specified. It has been demonstrated that the streamcluster scales well up to a count of 10 threads, but that after that point it begins to operate in an ineffective manner. The performance of swaptions and ferret is satisfactory up to 20 and 12 threads, respectively, but does not greatly improve after that. The region of Figure 2 depicted in Figure 1 shows the point at which these benchmarks stopped being able to grow as the number of processors rose. It is a commonly held belief that the number of threads in multi-threaded applications should correspond to the number of cores on multi-core processing systems in order to achieve the highest possible level of performance. However, this is not the case with the standards that are being used here. Because the execution time varies at random as shown in Figure 2, it is not possible to use any method or methodology to determine the thread count that results in the least amount of time spent executing code. In order to solve this problem, a one-of-a-kind searching optimization strategy has been presented for estimating the number of threads used in multi-threaded programs.

The following is a summary of the significant contributions that the paper makes:

- To investigate the usage of searching algorithms that are influenced by bio-inspired algorithms in order to find the optimal thread count.

- To determine the best possible number of threads and to speed up the processing time by employing an adaptive version of the Manta-Ray foraging optimization technique.

The remaining portions of the paper are organised as follows: There is a list of recent papers in section 3, and a discussion of the suggested method for determining an ideal number of threads based on the AMRFO can be found in section 4. In the fifth section, we analyse the results of the simulation, and in the last section, we address the conclusion as well as the possible uses of this research in the future.

3 Related Works

A learning-based technique was proposed by Agarwal et al. as an additional method for estimating application-related performance. This method was proposed in addition to varying degrees of parallelism. In order to evaluate the performance of the software when used in parallel applications, a p-threads library was utilised. In addition, low-level characteristics that signify a significant part of the work for the prediction of speed-ups were investigated. For multi-threaded Programs, the suggested method was carried out utilising the SPLASH-3.0 and PARSEC-3.0 programming languages. The Gaussian process regression and the linear regression both obtained higher levels of success after being compared to a number of different ML (machine learning) methodologies[2].

On multi-core CPUs, Sander et al. proposed a method known as RPPM, which stands for rapid performance prediction of multi-threaded workloads. On multi-core hardware, an automatic logical performance design was designed specifically for use with multi-threaded applications. Gathering microarchitecture-independent characteristics of a multi-threaded workload was one of the tasks that the RPPM performed in order to determine how well a previous concealed multi-core structure performed[27].

Tao Ju et al. came up with the idea of energy-effective thread mapping for heterogeneous multi-core systems and proposed that it could be accomplished by dynamically altering the count of the threads. The TCPM, which stands for "thread count prediction model," was used in the regression technique to determine the appropriate number of threads to use in light of the characteristics of heterogeneous multi-core structures and the behaviours of programs when they are executing. In addition, the dynamic predictive thread mapping (DPTM) algorithm was used. This algorithm uses a model of prediction to find the ideal count of threads and vigorously alters the number of energetic hardware threads based on the phase change of the program that is being executed in order to achieve the ideal energy effectiveness[37].

Jihyun Park et al. proposed using a method known as dynamic analysis (DA) to investigate the factors that lead to concurrency flaws in multi-process environments as well as the many types of bugs that can result from them. The hooking theory presents a potential method for lowering the percentage of false positives produced by the proposed system. The error that occurred between processes and threads was analysed with the help of the analysis of shared memory calls. The established system was carried out in an environment running Linux by utilising the Ewha COncurrency Detector, or ECO for short[14].

A multi-core and multi-thread-based optimization was presented by Xin Wei et al. in order to tackle the extremely large-scale version of the traveling salesman problem (TSP). The recommended method was carried out using the Delphi programming language in order to resolve the average and large-scale TSP occurrences from TSPLIB. This strategy has the potential to significantly speed up the penetration procedure without taking into account the loss in data quality[42].

A model-based optimization method that was effective in terms of the amount of energy it consumed was reported by Rauber et al. for use in multi-threaded applications. This method is mainly focused on the DVFS, also known as dynamic voltage and frequency scaling, which is a technique that may be utilised across a range of platforms. In order to gain an understanding of the energy efficiency, we made use of the deliberative models of analysis, in addition to performing the study itself. In addition to this, the potential impact that the number of threads simultaneously attempting to complete a multi-threaded program might have was also taken into consideration. In addition, the energy-delay product was covered, which enabled the power to be covered in relation to the square of the implementation time[28].

Demetrios et al. presented an energy consumption and performance trade-off for the parallel application on heterogeneous multi-processing systems in their article. The performance and energy trade-offs for the parallel application were suggested, in addition to a single instruction set. A whole new analytical design was designed in order to analyse the performance and power utilisation of the system. In this case, the parameters were closely matched by using some offline data that were properly sampled. These models were used over the entirety of the configuration space in order to compute the performance of the application as well as the energy utilisation. These offline predictions, which could be used to update the choice of the outline, were used to indicate the decision that was made regarding how the Pareto-optimal outlines of the model should be assessed[9].

Saez et al. made a suggestion regarding the loop-based OpenMP applications that were utilised in the logical and industrial areas. These applications were responsible for the development of the parallel benchmark suites, which were then utilised in the process of estimating the performance of many-core architectures. The traditional OpenMP approaches were unable of successfully handling load imbalance due to the scheduling of the loop in the program. In order to overcome these challenges, the method known as AID (Asymmetric Iteration Distribution) was conceived of over the course of this research and implemented on two distinct AM platforms in libgomp[4].

4 Proposed Methodology

There has been a lot of interest in developing ways for attaining maximum performance on multi-core architectures as a result of the widespread availability of multi-core computers. This interest is due to the fact that multi-core processors are now widely available. The performance of a multi-threaded program is determined by the total number of threads that are allowed to run on a multi-core structure. As a result, estimating the ideal thread count that will result in satisfactory performance is a considerable challenge. When it comes to multi-threaded applications running on multi-core computer systems, there have been established a lot of different approaches for forecasting the minimal count of threads. These techniques suffer from limited computer performance and poor accuracy in their predictions, which are two of their shortcomings. As a result, the purpose of this research is to present an optimal number of threads prediction model by making use of the MRFO[41] in order to boost the performance of computing while simultaneously reducing the amount of energy used.

The cognitive processes of manta rays served as a source of motivation for the development of the MRFO algorithm. Foraging techniques such as the chain, cyclone, and somersault are included in its three stages. Somersault foraging is given a fresh look with the introduction of a new algorithm based on Cauchy mutation in this work. The new AMRFO algorithm, which was created to boost exploitation and exploration capabilities, is the result of combining cyclone foraging, somersault foraging, and the chain foraging of MRFO.

The purpose of this exercise is to determine how long it takes for the multi-threaded apps to run by running them numerous times with a limited amount of data. Before it can be used, the programs need to be run on the hardware of your choice. The user and the algorithm work together to determine the total number of times that this procedure will be carried out. The AMRFO is a searching method that locates a new thread count with each iteration by using the positions of Manta Rays within the provided search space to determine the new count. After that, the application is executed using the newly produced number of threads, and the time it takes to complete is compared to the time it took during earlier runs. In the end, it will return the thread count that can be executed in the shortest amount of time.

The amount of data that is used in the experiments is extremely important because if there is too little data, then not all of the processors will be utilised, but if there is too much data, then the algorithm will take longer to process. The executions take a relatively short amount of time because the amount of data that is being inputted is so insignificant in contrast to its initial size. When selecting the amount of data to keep all processors active during the entirety of the operation, it is imperative that the number of cores that are present in the computer system be taken into consideration at all times.

4.1 Mathematical Models Representing Cognitive Activities of Manta Rays

At first, the Manta Ray population is initialized randomly as given by

$$X^d_{rand} = L^d_b + r(U^d_b - L^d_b) \tag{1}$$

Here,d is the dimension and the arbitrary location is defined as X in the hunt space, the lower and upper limit of the d dimension is defined as L and U respectively and a random number is denoted as r in the range of [0, 1].

4.1.1 Chain Foraging

During this stage, manta rays are able to determine the location of the plankton, allowing them to navigate in that general direction. Despite the fact that the AMRFO excellent solution is not defined, the algorithm determines that a high concentration indicates the good solution. Manta Rays migrate from their heads to their tails in order to construct a chain of foragers. The following constitutes the accurate model of chain foraging:

$$X^d_i(t{+}1) = \begin{cases} X^d_i(t) + r \times ((X^d_{best}(t) - X^d_i(t)) \\ +\alpha \times (X^d_{best}(t) - X^d_i(t)), i = 1 \\ \\ X^d_i(t) + r \times ((X^d_{i-1}(t) - X^d_i(t)) \\ +\alpha \times (X^d_{best}(t) - X^d_i(t)), i = 2, ..., N \end{cases} \tag{2}$$

Here, in d^{th} dimension the location of $(i-1)^{th}$ Manta Ray is denoted as $X^d_{(i-1)}(t)$ at time t and also the location of i^th Manta Ray is denoted as $X^d_i(t)$. A high concentration plankton is denoted as $X^d_{best}(t)$ and the constant is denoted as α that can be given as:

$$\alpha = 2 \times r \times \sqrt{\log} \tag{3}$$

4.1.2 Cyclone Foraging

This behaviour is scientifically designed by the subsequent expressions.

$$X^d_i(t{+}1) = \begin{cases} X^d_i(t) + r \times ((X^d_{best}(t) - X^d_i(t)) \\ +\beta \times (X^d_{best}(t) - X^d_i(t)), i = 1 \\ \\ X^d_i(t) + r \times ((X^d_{i-1}(t) - X^d_i(t)) \\ +\beta \times (X^d_{best}(t) - X^d_i(t)), i = 2, ..., N \end{cases} \tag{4}$$

$$\beta = 2\exp(r_1 \times ((T - t + 1)/T)) \times \sin(2\pi r) \tag{5}$$

The maximum amount of iterations that can be performed is indicated here as T, the weight factor is indicated as $beta$, and the random value that can fall anywhere between 0 and 1 is indicated as r_1. The searchers search in an arbitrary manner because they choose the locations of the prey as their reference points. This procedure provides an appropriate means of exploitation for the area where the good solution exists. In addition, the exploration process can be built in such a way as to improve its effectiveness by selecting any arbitrary site as the process's reference point.

4.1.3 Somersault Foraging

During this procedure, the individual position can be improved by being brought up to date in order to boost the local ability, which can be described as follows:

$$X^d_i(t+1) = X^d_i(t) + S \times (r_2 \times (X^d_{best} - r_3 \times X^d_i(t)),$$
$$i = 1, 2, ..., N \tag{6}$$

In this context, the somersault coefficient is referred to as S, and its value in this scenario is equal to 2. r_1

and r_2 are the definitions of the arbitrary numbers, and they are defined as being in the range of [0, 1]. Because the symbol S is a constant throughout this process, it does not improve upon itself and instead contributes to the formation of a local optimal point. As a result, the Cauchy mutation process has been implemented in order to enhance the capability of exploration while avoiding the use of the local search space.

The 1D Cauchy function can be defined as:

$$f(X) = 1/\pi \times 1/(X^2 + 1), -\infty < X < \infty \quad (7)$$

The PDF (probability density function) of Cauchy function is described as:

$$F(X) = 1/2 + (1/\pi)\arctan(X) \quad (8)$$

Cauchy's function can be thought of as the step size of mutation being developed. Once the individual reaches the point where the local optimality is reached, a higher step size facilitates the distinct jump out of the local optimality point. When an individual is looking for the optimal solution and is getting close to convergence, the speed of convergence can be increased by taking lower step sizes. The new scientific model of this process is defined as:

$$X_i^d(t+1) = X_i^d(t) + C \times (r_2 \times X_{best}^d(t) - r_3 \times X_i^d(t))$$
$$i = 1, 2, ..., N \quad (9)$$

In this context, the Cauchy distribution's arbitrary number is denoted by the letter C. The final step in completing MRFO and achieving AMRFO is to combine the new somersault foraging method with the chain and cyclone foraging strategies.

4.2 AMRFO Algorithm

The most important factor that has a major impact on the running performance of the program is determining the correct number of threads. In this part, the new and improved AMRFO algorithm for predicting the best number of threads to use in parallel processing is presented. The model demonstrates that there is a correlation between the amount of threads and the performance of the program. Chain foraging, cyclone foraging and somersault foraging based on the Cauchy mutation technique are the three stages that make up this algorithm. Because the present position of the Manta Ray determines the number of threads, the fitness function is evaluated for each of these positions, and the position that yields the best results is recorded as the optimal solution. The proposed optimal threads prediction model contains the following steps:

1. Set N, which stands for the number of Manta Rays, to a positive number.
2. Use the eq. (1) to figure out where is the position of each Manta Ray. Here, the positions of the Manta Rays are integers that represent the number of threads
3. Set the starting best position to the number of cores or processors that can be used to do calculations.
4. Do the next steps again and again until you reach the maximum number of iterations or find the answer you want.
   (a) For every Manta-Ray
      i. Determine a random number r between [0,1]; if it is less than 0.5, execute Cyclone foraging using Eq. (4) and update the new position.
      ii. Otherwise, execute chain foraging using Eq. (2) and update the new position.
      iii. Calculate the fitness value for the new position determined in the preceding steps.
      iv. If the current fitness value is smaller than the fitness value of the previously determined best position, make the new position of the Manta Ray the optimal position.
   (b) For each Manta-Ray, execute the somersault foraging eq. (9). Calculate the fitness value and identify the optimal place.
5. The optimal number of threads is represented by the best position attained in the preceding steps.

Figure 3 illustrates the AMRFO flowchart with all pertinent information. In this procedure, all Manta Rays are subjected to either Chain or Cyclone foraging, and the optimal position is chosen from among them. Then, utilising somersault foraging, a position that is superior to the previous best location is selected. This is repeated until the maximum number of iterations has been reached or an exit condition has been satisfied.

In this instance, the fitness value is the amount of time required to execute the program with the provided number of threads. following steps are performed to obtain fitness value for Manta Rays

1. Determine the present location of the Manta Ray.
2. Obtain the multi-threaded application and sample input data from which to ascertain the thread count.

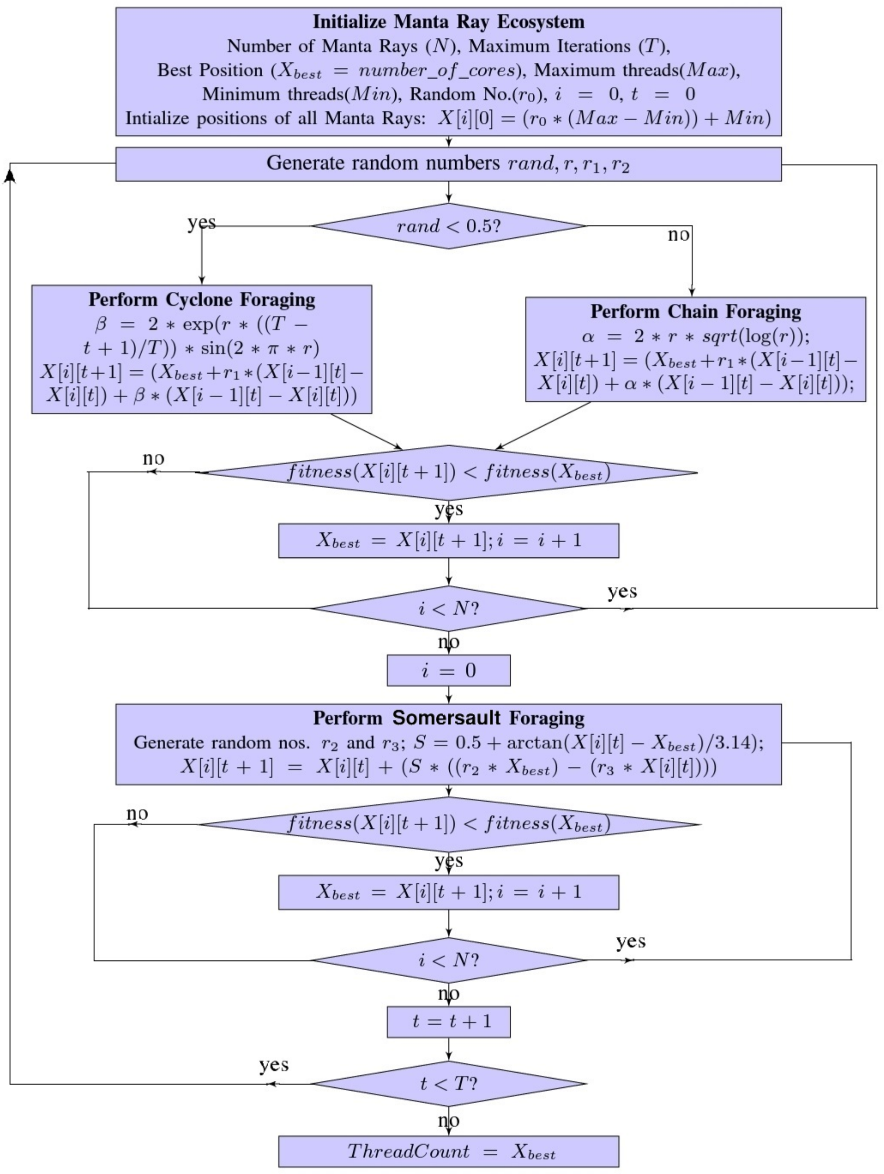
Initialize Manta Ray Ecosystem
Number of Manta Rays (N), Maximum Iterations (T),
Best Position ($X_{best} = number\_of\_cores$), Maximum threads($Max$),
Minimum threads(Min), Random No.(r_0), $i = 0$, $t = 0$
Intialize positions of all Manta Rays: $X[i][0] = (r_0 * (Max - Min)) + Min)$
Generate random numbers $rand, r, r_1, r_2$
$rand < 0.5$?
yes
no
Perform Cyclone Foraging
$\beta = 2 * \exp(r * ((T - t + 1)/T)) * \sin(2 * \pi * r)$
$X[i][t+1] = (X_{best} + r_1 * (X[i-1][t] - X[i][t]) + \beta * (X[i-1][t] - X[i][t]))$
Perform Chain Foraging
$\alpha = 2 * r * sqrt(\log(r))$;
$X[i][t+1] = (X_{best} + r_1 * (X[i-1][t] - X[i][t]) + \alpha * (X[i-1][t] - X[i][t]))$;
$fitness(X[i][t+1]) < fitness(X_{best})$
no
yes
$X_{best} = X[i][t+1]; i = i + 1$
$i < N$?
yes
no
$i = 0$
Perform Somersault Foraging
Generate random nos. r_2 and r_3; $S = 0.5 + \arctan(X[i][t] - X_{best})/3.14)$;
$X[i][t+1] = X[i][t] + (S * ((r_2 * X_{best}) - (r_3 * X[i][t])))$
$fitness(X[i][t+1]) < fitness(X_{best})$
no
yes
$X_{best} = X[i][t+1]; i = i + 1$
$i < N$?
yes
no
$t = t + 1$
$t < T$?
yes
no
$ThreadCount = X_{best}$

Figure 3: AMRFO flowchart

3. Execute the program with the same thread count as represented by the current location of Manta Ray.

4. The fitness value is the time required to complete a task.

5 Results and Discussions

The AMRFO approach is modelled around the widely used PARSEC benchmark suite. The PARSEC suite's six benchmarks $blackscholest$, $ferret$, $radiosity$, $swaptions$, $water\_nsquared$, and $water\_spatial$ are utilised. Our experimental system is a 2.2 GHz Intel Xeon E5-2698-v4 server, as shown in the table 1 . This system has 256GB of main memory and 40 logical cores. We chose to perform our study on Linux since it provides numerous resources for profiling and analysing software. We repeated each test ten times and averaged the data. There are six different types of datasets that each software must process in order to pass the PARSEC benchmark. AMRFO's fitness function

Table 1: Experimental Setup

$Server$	NVIDIA DGX Station
$Processor$	Intel Xeon-E5-2698v4 2.2 GHz
$PhysicalCores$	20
$LogicalCores$	40
$PrimaryMemory$	256GB
$OperatingSystem$	Linux

uses the smaller "simsmall" input dataset to determine the runtime.

Table 2 displays the results that were achieved for the streamcluster benchmark after each iteration that was run. The experiment uses a total of four Manta Rays across its four different iterations. The initial best position, denoted by X_{best}, is predetermined to be an integer number equal to the total number of processor cores present in the system, which for the purpose of this study is 40. The $Behaviour$ column displays the type of foraging strategy utilised by Manta Rays for computation, while the $Iteration$ column displays the number of times the calculation was performed. In order to determine the next location for the Manta Rays, the Cyclone and Chain foraging strategies employ the random values r_1 and r_2, while the Somersault foraging strategy uses the random number r_3. The value of X_{best} in each iteration is the best position that was determined by comparing the fitness values of all Manta Rays' prior best positions with their present positions. The columns labeled "Manta Ray," "Position," and "Fitness," respectively, provide information on the identification number, the number of threads, and the amount of time it took to execute. Following the completion of the cyclone foraging in the first iteration, which initialises the positions of all four manta rays in relation to the currently determined X best position, the somersault foraging is carried out. In later rounds, the Chain foraging takes the place of the Cyclone foraging that was previously done. The findings make it abundantly evident that the optimum number of threads is 12, which is a far lower value than the number of cores present in the system.

In order to evaluate the effectiveness of a parallel program, its speedup is a great metric to use. if a sequential program on a single core takes $T_s(1)$ seconds to finish, and a parallel program on P number of processors will take $T_p(P)$ seconds then the speedup, $S_p(P)$, is defined as follows:

$$S_p(P) = T_s(1)/T_p(P) \tag{10}$$

The speedup is computed for the same benchmark programs in order to measure the prediction accuracy of the AMRFO-based prediction method that was described before. As can be seen in Table 3, the projected speedup is evaluated in comparison to the system's best thread count for benchmark programs. The thread count acquired by utilising AMRFO is denoted as N here. The following is a definition of the improvement that the AMRFO offers over the conventional method:Â

$$\delta = (S_p(N) - S_p(40))/S_p(40) \tag{11}$$

It is clearly evident that the outcomes attained through the utilisation of AMRFO are noticeably superior than the outcomes attained through the utilisation of the assumption that the number of threads in a system is exactly the number of cores in the system. Figure 4 displays, in graphical form, the comparisons of accelerated times that were found in Table 3. When the number of threads is lower than forty, the $blackscholes$, $radiosity$, and $water\_nsquared$ algorithms provide more accurate results. On the other hand, the $ferret$, $swaptions$ and $water\_spatial$ had better outcomes with a thread count is more than 40. The $water\_spatial$ demonstrated a speedup gain of 300% as compared to when it was operating with 40 threads. The $ferret$ and $water\_nsquared$ benchmarks all exhibit considerable speedups, which demonstrates the usefulness of the approach.

The difference in execution durations between using a single thread and the optimal number of threads is displayed in Figure 5. The amount of time needed to complete the $water\_spatial$ and $water\_nsquared$ algorithms have been elongated so that it will appropriately fit within the graph. Every single one of the benchmarks demonstrates a considerable increase in speed.

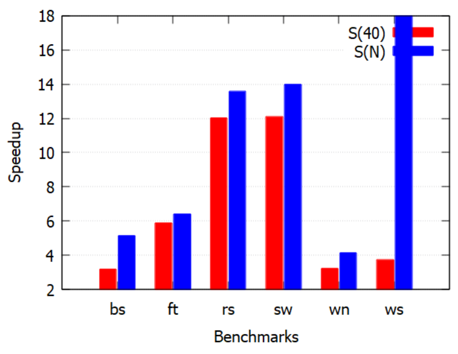

Figure 4: S(40) vs S(N)

Table 2: Streamcluster benchmark: Calculations of positions and fitness values

$Sr.$	$Iteration$	$Behaviour$	$r1$	$r2$	$r3$	$MantaRay$	$Position$	$Fitness$	X_{best}
0	init						40	0.521	40
1	1	Cyclone	0.707228722			1	55	10.888	
2		Cyclone	0.169801442			2	39	1.605	
3		Cyclone	0.066163426			3	38	1.775	
4		Cyclone	0.902467464			4	26	1.1	
5		somersault		0.762858734	0.13506918	1	73	15.207	
6		somersault		0.88273134	0.425764812	2	45	8.332	
7		somersault		0.611535392	0.025749147	3	39	2.074	
8		somersault		0.058743301	0.685183239	4	12	0.691	12
9	2	Chain	8.52E-04			1	12	0.711	
10		Chain	0.945285619			2	51	10.434	
11		Chain	0.788572721			3	21	0.955	
12		Chain	0.104572951			4	23	1.135	
13		somersault		0.802563399	0.577423679	1	41	6.335	
14		somersault		0.781250609	0.886794933	2	15	0.774	
15		somersault		0.433626861	0.701207178	3	17	0.784	
16		somersault		0.996348733	0.872226209	4	13	0.725	12
17	3	Chain	0.008204582			1	11	0.71	
18		Chain	0.668251891			2	51	10.086	
19		Chain	0.139353127			3	11	0.707	
20		Chain	0.811923842			4	18	0.867	
21		somersault		0.666769548	0.075264064	1	46	8.742	
22		somersault		0.424714497	0.73875491	2	12	0.691	
23		somersault		0.764504423	0.938892559	3	12	0.691	
24		somersault		0.299860293	0.930062153	4	12	0.691	12
25	4	Chain	0.003367132			1	11	0.707	
26		Chain	0.230433012			2	39	1.605	
27		Chain	0.113401663			3	12	0.691	
28		Chain	0.479832443			4	12	0.691	
29		somersault		0.65248563	0.975373003	1	12	0.691	
30		somersault		0.987842571	0.102818573	2	22	0.945	
31		somersault		0.90180858	0.762232542	3	14	0.706	
32		somersault		0.011032869	0.87732143	4	12	0.691	12

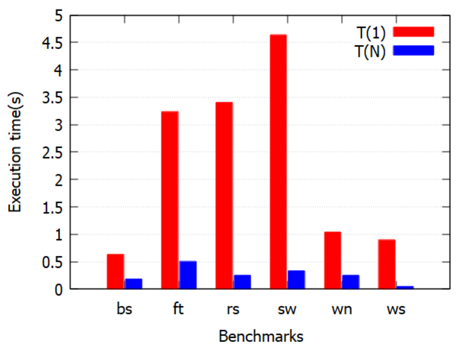

Figure 5: T(1) vs T(N)

6 Conclusion and Future Scope

In this article, an effective model for predicting threads is constructed by making use of the AMRFO algorithm. The prediction model makes it simple to ascertain the ideal number of threads for achieving the greatest possible speedups. The findings from the simulation demonstrate that the proposed algorithm makes effective use of the limited search space and soon arrives at a solution that is optimal. With the Cauchy mutation included, the algorithm was able to investigate a larger search space without becoming mired in a series of local optimum solutions. The approach that is given in this paper for determining the number of threads is an easy one to use, but it does have some downsides. For example, the user must first test the program with a very little amount of data before testing it with the actual input data. These overhead costs become negligible if the actual amount of data being inputted is a very large quantity. As a consequence of this, it is essential to make an informed decision regarding the data size prior to applying AMRFO to do an evaluation of an application. In the future, it will be possible to extend this method by making use of the various deep learning structures and also applying the method to mixed-mode workloads in order to improve its overall performance.

Table 3: Comparison of speedups between optimal thread count obtained with AMRFO and the system best thread count i.e. 40

$Benchmark$	$T_s(1)$	$T_p(40)$	$S_p(40) = T_s(1)/T_p(40)$	N	$T_p(N)$	$S_p(N) = T_s(1)/T_p(N)$	δ
blackscholes (bs)	0.636	0.198	3.21	39	0.183	5.17	8.27%
ferret (ft)	3.239	0.547	5.9	49	0.505	6.414	30%
radiosity (rs)	3.407	0.283	12.04	39	0.251	13.573	12.7%
swaptions (sw)	4.636	0.383	12.104	46	0.331	14.0	16.5%
water_nsquared (wn)	0.104	0.032	3.25	13	0.025	4.16	28%
water_spatial (ws)	0.09	0.024	3.75	72	0.005	18	300%

References

[1] Abeydeera, M., Subramanian, S., Jeffrey, M. C., Emer, J., and Sanchez, D. Sam: Optimizing multithreaded cores for speculative parallelism. *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 26:64–78, September 2017.

[2] Agarwal, N., Jain, T., and Zahran, M. Performance prediction for multi-threaded applications. *In International Workshop on AI-assisted Design for Architecture*, June 2019.

[3] Alexey, P. and Shichkina, Y. Algorithms for optimization of processor and memory affinity for remote core locking synchronization in multithreaded applications. *Information (Switzerland)*, 9(21):957–963, January 2018.

[4] Carlos, S. J., Castro, F., and Prieto-Matias, M. Enabling performance portability of data-parallel openmp applications on asymmetric multicore processors. *49th International Conference on Parallel Processing-ICPP*, pages 1–11, August 2020.

[5] ChavarrÃa-Miranda, D., Huang, Z., and Chen, Y. High-performance computing (hpc): Application & use in the power grids. *2012 IEEE Power and Energy Society General Meeting*, pages 1–7, July 2012.

[6] de Almeida, F. L., Rosa, R. L., and Rodríguez, D. Z. Voice quality assessment in communication services using deep learning. In *2018 15th International Symposium on Wireless Communication Systems (ISWCS)*, pages 1–6. IEEE, 2018.

[7] de Supinski, B. R., Scogland, T. R. W., Duran, A., Klemm, M., Bellido, S. M., Olivier, S. L., Terboven, C., and Mattson, T. G. The ongoing evolution of openmp. *Proceedings of the IEEE*, 106(11), November 2018.

[8] Deep, K., Arya, M., and Barak, S. A new multi-swarm particle swarm optimization and its application to lennard-jones problem. *INFOCOMP Journal of Computer Science*, 9(3):52–60, Sep. 2010.

[9] Demetrios, C. and Georgiou, K. Performance and energy trade-offs for parallel applications on heterogeneous multi-processing systems. *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 232–233, October 2019.

[10] Donno, D. D., Esposito, A., Tarricone, L., and Catarinucci, L. Introduction to gpu computing and cuda programming: A case study on fold. *IEEE Antennas and Propagation Magazine*, 52(3), June 2010.

[11] Furat, A.-O., Asad, A., and Mohammadi, F. A. A power-aware hybrid cache for chip-multi processors based on neural network prediction technique. *International Journal of Parallel Programming*, 49(1):1–21, June 2021.

[12] Gupta, O. and Kahlon, K. S. Performance evaluation of network based distributed supercomputing environment. *INFOCOMP Journal of Computer Science*, 6(3):15–18, Sep. 2007.

[13] Hamid, S. H. A., Nasir, M. H. N. M., Ming, W. Y., and Hassan, H. Multi-threading and shared-memory pool techniques for authorization of credit card systems using java. *INFOCOMP Journal of Computer Science*, 7(3):60–69, Sep. 2008.

[14] Jihyun, P., Choi, B., and Jang, S. Dynamic analysis method for concurrency bugs in multi-process/multi-thread environments. *International Journal of Parallel Programming*, 48(6):1032–1060, December 2020.

[15] Jordane da Silva, M., Carrillo Melgarejo, D., Lopes Rosa, R., and Zegarra Rodríguez, D.

Speech quality classifier model based on dbn that considers atmospheric phenomena. *Journal of Communications Software and Systems*, 16(1):75–84, 2020.

[16] Langmead, B., Wilks, C., Antonescu, V., and Charles, R. Scaling read aligners to hundreds of threads on general-purpose processors. *Bioinformatics*, 35(3):421–432, February 2019.

[17] Madonna, S. J. and Venkatesh, T. G. Analytical derivation of concurrent reuse distance profile for multi-threaded application running on chip multi-processor. *IEEE Transactions on Parallel and Distributed Systems*, 30(8):1704–1721, August 2019.

[18] Mauro, O., Cheikh, A., Cerutti, G., and Menichelli, A. M. F. Investigation on the optimal pipeline organization in risc-v multi-threaded soft processor cores. *2017 New Generation of CAS (NGCAS)*, pages 45–48, September 2017.

[19] Mehrdad, G. and Babamir, S. M. Runtime deadlock tracking and prevention of concurrent multithreaded programs: A learning based approach. *Concurrency and Computation: Practice and Experience*, 32(10), June 2020.

[20] Moore, R. W., Childers, B. R., and Xue, J. Performance modeling of multithreaded programs for mobile asymmetric chip multiprocessors. *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 957–963, August 2015.

[21] Nagasakaa, Y., Matsuoka, S., Azad, A., and Buluc, A. Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors. *Parallel Computing*, 90, December 2019.

[22] Nunes, R. D., Pereira, C. H., Rosa, R. L., and Rodríguez, D. Z. Real-time evaluation of speech quality in mobile communication services. In *2016 IEEE International Conference on Consumer Electronics (ICCE)*, pages 389–390. IEEE, 2016.

[23] Nunes, R. D., Rosa, R. L., and Rodríguez, D. Z. Performance improvement of a non-intrusive voice quality metric in lossy networks. *IET Communications*, 13(20):3401–3408, 2019.

[24] Palafox-Maestre, L. E. and Garcia-Macias, J. A. A bio-inspired approach for data dissemination in wireless sensor networks. *INFOCOMP Journal of Computer Science*, 5:19–27, Sep. 2006.

[25] Paul-Jules, M., Smith, A., and Dubach, C. A machine learning approach to mapping streaming workloads to dynamic multicore processors. *In Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*, pages 113–122, June 2016.

[26] Paznikov, A. Optimization of remote core locking synchronization in multithreaded programs for multicore computer systems. *Information Technology in Industry*, 6(2):7–12, 2018.

[27] Pestel, S. D., den Steen, S. V., Akram, S., and Eeckhout, L. Rppm: Rapid performance prediction of multithreaded applications on multicore hardware. *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 183–186, March 2018.

[28] Rauber, T., RÃ$\frac{1}{4}$nger, G., and Stachowski, M. Model-based optimization of the energy efficiency of multi-threaded applications. *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*, pages 44–61, October 2019.

[29] Rinard, M. Analysis of multithreaded programs. *International Static Analysis Symposium*, 2126:1–19, July 2001.

[30] Rodríguez, D. Z., Carrillo, D., Ramírez, M. A., Nardelli, P. H. J., and Möller, S. Incorporating wireless communication parameters into the e-model algorithm. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 29:956–968, 2021.

[31] Rodríguez, D. Z., da Silva, M. J., Silva, F. J. M., and Junior, L. C. B. Assessment of transmitted speech signal degradations in rician and rayleigh channel models. *INFOCOMP Journal of Computer Science*, 17(2):23–31, 2018.

[32] Rodríguez, D. Z. and Junior, L. C. B. Determining a non-intrusive voice quality model using machine learning and signal analysis in time. *INFOCOMP Journal of Computer Science*, 18(2), 2019.

[33] Rodríguez, D. Z., Rosa, R. L., Almeida, F. L., Mittag, G., and Möller, S. Speech quality assessment in wireless communications with mimo systems

using a parametric model. *IEEE Access*, 7:35719–35730, 2019.

[34] Rodríguez, D. Z., Rosa, R. L., and Bressan, G. A billing system model for voice call service in cellular networks based on voice quality. In *2013 IEEE International Symposium on Consumer Electronics (ISCE)*, pages 89–90. IEEE, 2013.

[35] Sayadi, Hossein, and Homayoun, H. Scheduling multithreaded applications onto heterogeneous composite cores architecture. *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*, IEEE, October 2017.

[36] Solmaz, S. and Lu, L. Memory bandwidth prediction for hpc applications in numa architecture. *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 1115–1122, August 2019.

[37] Tao, J., Zhang, Y., Zhang, X., Du, X., and Dong, X. Energy-efficient thread mapping for heterogeneous many-core systems via dynamically adjusting the thread count. *Energies*, 12(7), April 2019.

[38] Terra Vieira, S., Lopes Rosa, R., Zegarra Rodríguez, D., Arjona Ramírez, M., Saadi, M., and Wuttisittikulkij, L. Q-meter: Quality monitoring system for telecommunication services based on sentiment analysis using deep learning. *Sensors*, 21(5):1880, 2021.

[39] Turakhia, Yatish, Liu, G., and Marculescu, S. G. D. Thread progress equalization: Dynamically adaptive power-constrained performance optimization of multi-threaded applications. *IEEE Transactions on Computers*, 66(4):731–744, April 2017.

[40] Vieira, S. T., Rosa, R. L., and Rodríguez, D. Z. A speech quality classifier based on tree-cnn algorithm that considers network degradations. *Journal of Communications Software and Systems*, 16(2):180–187, 2020.

[41] Weiguo, Z., Zhang, Z., and Wang, L. Manta ray foraging optimization: An effective bio-inspired optimizer for engineering applications. *Engineering Applications of Artificial Intelligence*, 87, August 2020.

[42] Xin, W., Ma, L., Zhang, H., and Liu, Y. Multi-core-, multi-thread-based optimization algorithm for large-scale traveling salesman problem. *Alexandria Engineering Journal*, 60(1):189–197, February 2021.

[43] Yun, J., Park, J., and Baek, W. Hars: A heterogeneity-aware runtime system for self-adaptive multithreaded applications. *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 52, June 2015.