International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING

ISSN:2147-6799 www.ijisae.org Original Research Paper

Reinforced Manta Ray Foraging Optimiser for Determining the Optimal Number of Threads in Multithreaded Applications

S H Malave[1], S K Shinde[2]

Submitted: 08/09/2022 **Accepted:** 26/12/2022

Abstract: Thread management affects operating system factors, resulting in execution time delays. These factors may improve or degrade depending on the design of the program and the number of threads. Therefore, for any multithreaded application, changes in these factors indicate whether the selected thread count is appropriate or not. This paper proposes a method that combines manta ray foraging optimisation and the Thread-reinforcer algorithm. The new algorithm predicts the best thread count by using three manta ray foraging strategies: chain, cyclone, and somersault; however, it selects the thread count with the highest fitness value as the best solution. The fitness function computes the fitness value by analysing OS factors such as CPU utilisation, context switching rate, CPU migration rate, page fault rate, and execution time. The multithreaded applications are run multiple times with a small data size to collect values for these factors. We tested the proposed work on fifteen programs in the PARSEC 3.0 benchmark suit. The results show that the optimal thread count for seven programs is greater than the number of processors and equal to the number of processors for the remaining eight programs. This study also demonstrates that the proposed approach takes less time to determine the solution than the Thread-reinforcer.

Keywords: *Parallel programs, threads, optimisation, nature-inspired.*

1. Introduction

In high-performance computing, parallel programs try to use most of the cores available in the machine to complete the execution of tasks efficiently. The programmers suggest the optimal number of threads before executing the program on the target machine. They devote a significant amount of time to understanding the issues in parallel programs and determining the count of threads for which the program can take less time to execute. The multithreaded applications are designed to take advantage of high-performance computing platforms to execute complex programs efficiently. In today's world, computers ranging from desktops to mainframes use multiprocessor CPUs. Many applications are developed using various parallel programming languages and libraries to solve complex real-world problems [1][2][3][4].

Multithreading is a technique in which an application can have several threads that can run on different processors at the same time [5]. The programmers should divide the code into small independent sections and allocate them to separate threads for execution. They mostly use a fork-join model [6] to convert the serial programs into parallel programs. In the fork-join model, a section of the program where instructions can be executed independently is searched, and then threads are created at runtime using programming libraries like OpenMP. The threads are terminated as soon as they finish the execution of their assigned sections except for the main thread, which continues to execute the remainder of the program.

Researchers have developed many tools [7][8] to detect parallelism in serial programs, but these tools do not help to determine how many threads are necessary at runtime. The general rule is to create threads in proportion to the number of processors in the system. This is true for a few CPU-intensive applications but does not apply to memory and IO-intensive applications. As a result, programmers must perform some manual work and rely on profiling tools to determine the number of threads. If they choose the wrong number, it can have a negative impact on the execution time. Thread management affects Operating System (OS) level factors such as shared memory, semaphores locks, scheduling time, waiting time, CPU migration, instructions per second, context switching, page fault, CPU utilisation, and so on [9][10]. It adds the overheads associated with these factors during the execution. These factors may reveal the internal working of the threads and help the programmer

[1]*Research Scholar*
Lokmanya Tilak College of Engineering, Koparkhirane, Navi Mumbai
sachinmalave@gmail.com
[2]*Professor*
Lokmanya Tilak College of Engineering, Koparkhirane, Navi Mumbai
skshinde@rediffmail.com
[1] *Lokmanya Tilak College of Engineering, Navi Mumbai-400709, INDIA*
ORCID ID : 0000-0002-1731-7536
[2] *Lokmanya Tilak College of Engineering, Navi Mumbai-400709, INDIA*
ORCID ID : 0000-0002-6709-3083

understand their behaviour. Furthermore, until the programmer is sure of the number of threads required for the given application to run on the target hardware, no more threads than the number of cores should be tried. As a result, parallel programmers need a framework or technique that can analyse the activities of threads in multithreaded programs and recommend the optimal thread count.

The CPU utilisation provides details about how much of the CPU is used by the running program during its execution. A decrease in this factor indicates that the threads are not fully utilising the available CPUs and are spending more time in the waiting state than in the running state.

The operating system uses context switching to suspend thread execution and assign its processor to another thread. The earlier thread must wait in a ready queue while the other one is running in the system. Context switching is useful in systems where multiple tasks must be performed concurrently, but it has a negative impact on parallel programming as it adds the waiting time to the total execution time.

CPU migrations occur when a thread is ready and a CPU other than the one where it was scheduled previously is available for execution. In this case, OS schedules the thread on the new CPU and resumes its execution. It causes the OS to reload the program and its data into the cache and memory close to the allocated CPU. This adds unnecessary delay to the program and increases the total execution time. It is observed that as the number of threads exceeds the number of cores, the migration rate begins to increase.

The page fault usually causes an exception, which is used to inform the operating system that the 'pages' from memory space must be loaded to continue execution. The program resumes normal function once all of the contents have been loaded into physical memory. This generally happens in the background and does not affect the normal execution. However, a rapid increase in page faults may indicate that the program is not behaving properly and prolong the execution time due to a lack of data or code inside the memory.

Determining optimal thread count is an optimisation problem that can be solved using optimisation algorithms. The researchers have developed many nature-inspired algorithms to solve problems in the engineering domain, such as ant colony, particle swarm, manta ray foraging, symbiotic search, bird-swarm and so on [11][12][13][14][15] manta ray foraging optimisation is a technique that simulates the behaviour of manta rays. Thus, rather than wasting time analysing the OS factors for the programs manually, it is recommended to use optimisation algorithms based on nature-inspired techniques. These algorithms can find the best solution in a known search space by mimicking the nature of organisms. The organisms travel to various locations and attempt to arrive at the best point as quickly as possible [16].

2. Motivations and Problem Statement

When we run a program with a single thread, it gets executed serially on a single processor. When we run it with two threads, we reduce the execution time by half by dividing the task into two equal sub-tasks. Thus, it is possible to reduce execution time by increasing the number of threads. Figure 1, Figure 2, and Figure 3 show the execution time taken by the three PARSEC benchmark [17] programs: streamcluster, ferret, and swaptions. We executed these programs at full load on a 12-core machine and recorded their execution time. The sub-figure (a) depicts the execution time for thread counts ranging from 2 to 24, whereas the sub-figure (b) depicts the zoomed-in portion of sub-figure (a) where the programs appear to be non-scalable.

Figure 1 (a) and Figure 1 (b) show that the streamcluster scales well up to 12 threads. After that, when the number of threads exceeds the number of cores, the execution time increases dramatically, demonstrating that the user should not run this program with more threads than the number of processors available in the system.

In Figure 2 (a) and Figure 2 (b), we can observe that beyond 12 threads, the performance of the ferret remains consistent, but the lowest execution time is recorded at 14 threads. However, in this case, even if the user decides to run the program for more threads than the number of cores, it does not affect the execution time.

Figure 3 (a) and Figure 3 (b) show that the swaptions have the shortest execution time with 16 threads, which is not the number of cores in the system. As a result, if the programmer decides to run it with 12 threads, it will take longer to complete than if it is run with 16 threads. As a result, there are programs with optimal thread counts that exceed the number of processors.

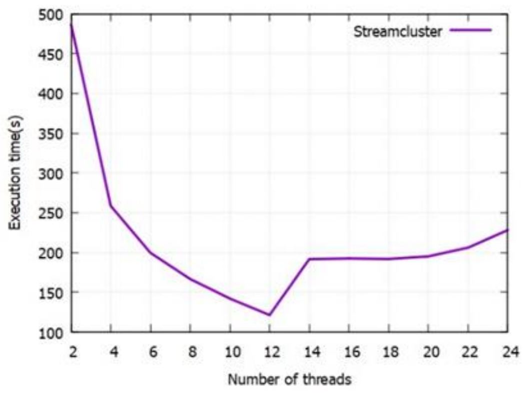

Figure 1 (a) Streamcluster benchmark execution time with thread counts ranging from 2 to 24.

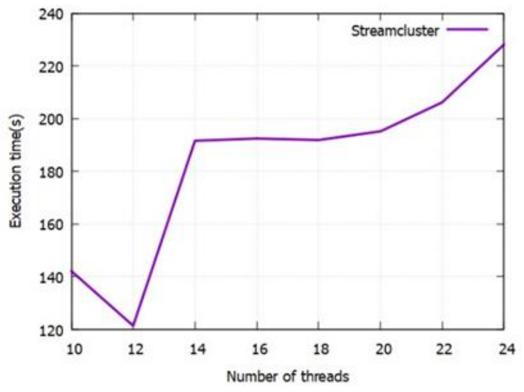

Figure 1 (b) Streamcluster benchmark execution time with thread counts ranging from 10 to 24.

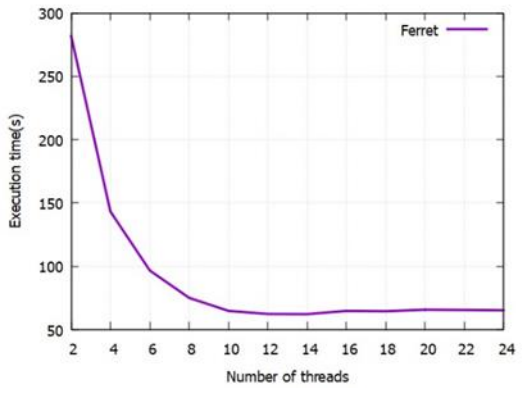

Figure 2 (a) Ferret benchmark execution time with thread counts ranging from 2 to 24.

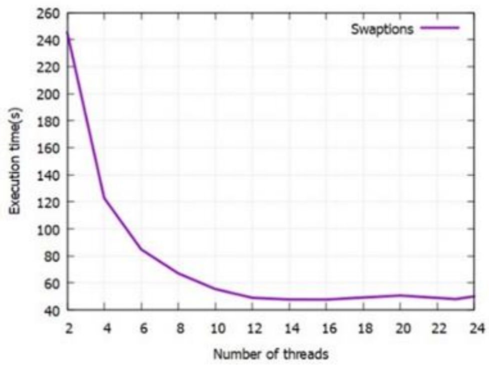

Figure 3 (a) Swaptions benchmark execution time with thread counts ranging from 2 to 24.

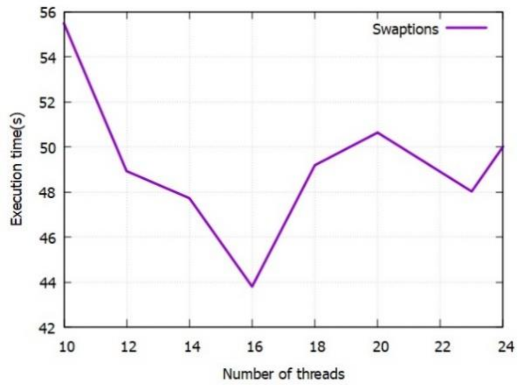

Figure 3 (b). Swaptions benchmark execution time with thread counts ranging from 10 to 24.

The execution time depends on how OS-level factors respond to running threads. Therefore, monitoring these factors is essential for finding the optimal count. One needs to change the thread count in each run and monitor activities to obtain these factors. If the method is applied manually, the user will undoubtedly lose a significant amount of time repeating the executions actively. Therefore, if a computer algorithm is developed to analyse these factors and predict the number of threads, the user can find the optimal thread count for multithreaded applications without wasting time.

3. Related Work

Pusukuri, et al [18] developed a simple technique called thread reinforcer for proactively calculating the required number of threads without redesigning the program or changing Operating System rules. Since calculating the proper number of threads for a multithreaded program periodically is a difficult task. In their study, a few OS factors are studied to get optimal thread count. Furthermore, architectural specifications such as memory management issues are not considered here.

Qin et al [19]. has presented a technique that delivers both fast response time and throughput for programs with threads. Based on the real-time system’s load, the running programs decide how many cores they require for execution. In this technique, the mapping between running threads and processor is stored properly which makes it to easily manage and control the threads and cores in the system. A central core arbiter manages the CPU allocations for the threads of all the programs running the system.

A technique called RPPM (rapid performance prediction of multithreaded workloads) was proposed in [20] on multicore processors. For multi-threaded applications, an automatic logical performance design was developed on multicore hardware. To detect performance on multicore platforms, microarchitecture-independent features are obtained for the multi-threaded workload.

AbdurRouf, et al [21] analyse the distribution of multiple threads on available processors. The OpenMP style parallel programming API is being used to create and spawn all the threads. The performance parameters are checked in single as well as multithreaded applications. They found in their study that increasing the number of threads proportional to the number of processors reduces program execution time on different multicore architectures. Similarly, their studies suggest that performance in these programs improves when thread allocation is done properly. The thread count should depend on the number of processors and cores in the machine, and the programmer should spawn threads equal to the processors available for use.

An energy-efficient model based on optimisation techniques was introduced in [22] for parallel applications. This proposed method concentrates on the DVFS (dynamic voltage and frequency scaling) which can be for many platforms. To capture the energy efficacy, mathematical models were designed. Moreover, the effect of the number of threads performing a multithreaded application was studied.

4. Proposed Methodology

Manta rays forage in groups and try to gather the maximum food [13]. They choose a location systematically that has more concentration of food. This type of behaviour can be simulated to develop an algorithm for finding the best solution in the available search space. The algorithm should use a fixed number of manta rays to move around in the search space and when a new position is discovered, it should be evaluated for suitability. As a result, a fitness function is also required. Here, the fitness function runs the application for the recommended number of threads and records the values for the various OS-level factors. These factors may improve or deteriorate depending on how the operating system reacts to the program.

The Reinforced Manta Ray Foraging Optimization (RMRFO) algorithm presented in this paper works in two steps. In the first step, the Thread count is obtained using manta ray foraging. In a second step, the program is run for a short period or with a limited amount of data by the fitness function, as described in Thread-reinforcer\cite{ct13}. During this step, the OS-level factors are analysed to determine whether the selected thread count is optimal. Then the application is executed with a full load with the thread count found in step two.

The amount of data used during experiments is important since too little data will not use all of the processors, but too much data will cause the algorithm to take longer to process. Because the input data is so little in comparison to its original size, the executions take relatively little time. The amount of data should be chosen in such a way that it should keep all processors busy during executions.

MRFO has three stages: chain, cyclone, and somersault foraging. The following sections explain the mathematical representations of these stages, fitness function and algorithms used in RMRFO.

4.1. Intialisation

At first, the count of the manta rays population and their starting positions in the solution search space are initialised. Here positions of these manta rays indicate the number of threads.

$$X = L_b^d + R\left(U_b^d - L_b^d\right) \tag{1}$$

In the search space, the location of the manta ray is denoted as X. L and U represent the lower and upper limits, and a random variable, R is defined in the range [0, 1].

4.2. Chain Foraging

Manta rays detect the location of plankton in this step and travel in their direction. Even though the good solution of RMRFO is not defined, the algorithm considers the position with high concentration of food as the good solution. Manta rays travel from head to tail to form a foraging chain. The mathematical formula for chain foraging is represented by :

$$X(t+1) = \begin{cases} X_i^d(t) + R_1\begin{pmatrix}\left(X_{best}^d(t) - X_i^d(t)\right) + \\ \alpha(X_{best}^d(t) - X_i^d(t))\end{pmatrix}, \\ \qquad i = 1 \\ X_i^d(t) + R_1\begin{pmatrix}\left(X_{i-1}^d(t) - X_i^d(t)\right) + \\ \alpha\left(X_{best}^d(t) - X_i^d(t)\right)\end{pmatrix}, \\ \qquad i = 2, \ldots., N \end{cases} \tag{2}$$

Here, the location of (i-1)th manta rays is denoted as Xi-1(t) and also the location of ith manta ray is denoted as Xi(t). R1 represents a random number in the range [0,1] and the high concentration place of food is denoted as Xbest(t) and the constant is denoted as α that can be given as:

$$\alpha = 2R\sqrt{|log(R1)|} \tag{3}$$

4.3. Cyclone Foraging

Following equation represents the cyclone foraging.

$$X_i^d(t+1) = \begin{cases} X_{best}^d + R_1\begin{pmatrix}\left(X_{best}^d(t) - X_i^d(t)\right) + \\ \beta\left(X_{best}^d(t) - X_i^d(t)\right)\end{pmatrix}, \\ \qquad i = 1 \\ x_{best}^d + R_1\begin{pmatrix}\left(X_{i-1}^d(t) - X_i^d(t)\right) + \\ \beta\left(X_{best}^d(t) - X_i^d(t)\right)\end{pmatrix}, \\ \qquad i = 2, \ldots, N \end{cases} \tag{4}$$

$$\beta = 2\,exp\left(R_1 \times \left(\frac{T-t+1}{T}\right)\right) \times sin(2\pi R_1) \tag{5}$$

Here, the maximum number of iterations is denoted as T, the weight factor is denoted as beta and the random number is denoted as R1 in the range of [0, 1].

Here, this process offers suitable exploitation to the good solution region. Further, to enhance the exploration process, this process can be designed by taking an arbitrary location as the reference location.

4.4. Somersault Foraging

In this process, the individual position can be updated to enhance the local ability which can be given as:

$$X_i^d(t+1) = \mathrm{X}(t) + S\left(R_2 X_{best}^d - R_3 X_i^d(t)\right), \tag{6}$$

$$where\ i = 1,2, \ldots., N$$

Here, the somersault coefficient is denoted as S which in this case is 2. The arbitrary numbers are defined as R2 and R3 that lie in the range of [0, 1]

4.5. RMRFO Algorithm

The proposed optimal threads prediction model has the following steps.

1. N = number of manta rays.
2. Calculate random positions for all the manta rays using Eq. (1).
3. X_{best} = number of cores.
4. Repeat the following steps until the maximum number

of iterations are completed or the optimal solution is found.

```
a. X(t) = current position of manta ray; R = random number between range [0,1].
b. For each manta ray
    i. If (R< 0.5) then perform cyclone foraging using Eq. (4), else perform chain foraging using Eq. (2).
    ii. X(t+1) = new position of manta ray.
    iii. fitness_value = fitness_function (X(t+1)).
    iv. If (fitness_value = 1) then $X_{best}$ = X(t+1).
c. For each manta ray
    i. Perform the somersault foraging using Eq. (6).
    ii. X(t+1) = new position of manta ray.
    iii. fitness_value = fitness_function (X(t+1)).
    iv. If (fitness_value = 1) then $X_{best}$ = X(t+1).
5. Optimal thread count = $X_{best}$.
```

The X_{best} variable holds the optimal thread count. For every manta ray, the algorithm executes cyclon foraging at the start and chain foraging in later iterations. The fitness function determines whether the newly calculated position X(t+1) is a better solution and, if so, assigns its value to X_{best}. The algorithm then executes somersault foraging for all of the manta rays. This method is repeated until all iterations have been completed or the exit condition has been met.

4.6. The fitness function

In this study, three PARSEC programs, ferret, swaptions, and streamcluster are chosen to analyse and record the values of OS factors. These programs are executed with a limited number of inputs. In this algorithm five factors namely CPU utilisation(CU), context switching rate(CS), CPU migration rate(CM) and page faults(PF) and execution time(ET) were studied to identify various upper and lower thresholds. The thresholds are the values from where the performance of these algorithms started to cease. Therefore, a software tool that can collect information related to various OS factors is required. The perf is a reliable tool for analysing multithreaded applications and multi-core system performance on Linux. It provides several useful command-line options for monitoring hardware counters across all processors. It collects information about the activities inside the processor so that the user can monitor and record the performance of the running program. The perf tool provides r migrations, page faults, cycles used, instructions per second, branches, and branch misses. These are extremely useful parameters that have been discussed previously and are being used here to see how they relate to thread count.eal-time values of CPU utilisation, context switches, CPU

Table 1. Effect if thread count on OS-level factors for streamcluster program.

Thread count	CU	CW (M/sec)	CM (K/sec)	PF (M/sec)	ET (sec)
6	2.056	0.004	0.128	0.033	1.6807
7	2.283	0.004	0.113	0.03	1.6994
10	3.54	0.005	0.069	0.017	1.8912
12	5.307	0.004	0.139	0.008	2.5849
13	7.927	0.004	0.233	0.002	7.115
16	7.715	0.005	0.001	0.002	7.4923
17	7.618	0.005	0.001	0.002	9.224
20	7.793	0.005	0.001	0.001	10.1678
21	7.762	0.005	0.002	0.001	11.9304
23	8.088	0.005	0.001	0.001	12.0144
24	8.326	0.005	0.001	0.001	11.7726
27	8.26	0.005	0.002	0.968	14.3681
32	8.34	0.005	0.002	0.807	17.1076

The PARSEC has included four types of input datasets: simsmall, simmedium, simlarge and native. The 'simsmall' is the smallest dataset and the 'native' is the largest dataset in the group. To decide various thresholds and exit conditions the selected three programs are executed on the target hardware with 'simlarge' input types. The programs are executed on a 12-core machine and the OS-level factors are noted using the perf tool. The effect of thread count on OS factors for the streamcluster program is shown in Table 1. It can be seen that as the number of threads increases, the CPU utilisation also increases. It can be also observed that context switching rate and CPU migration rate, show a significant change when the number of threads exceeds 12. The context switching rate is 0.001, which indicates that either no context switching occurred during the execution or the operating system was unable to schedule the threads waiting in the waiting state. In this case, the overheads of locks are so high that the program is almost come to a halt, resulting in a rapid increase in execution time. At the same time, the CPU migration rate is reduced to 0.001, indicating that almost no threads are migrated to other CPUs. Therefore, if the context switching rate and CPU migration rate both fall below 0.001, the execution time increases rapidly, and the programmer should not set the number of threads greater than this point.

Table 2. Effect if thread count on OS-level factors for ferret program.

Thread count	CU	CW (M/sec)	CM (K/sec)	PF (M/sec)	ET (sec)
6	5.102	0.421	0.093	0.013	2.9665
7	5.149	0.424	0.1	0.013	2.9688
10	5.07	0.431	0.112	0.013	3.0584
12	5.121	0.459	0.117	0.013	3.0671
13	5.122	0.462	0.117	0.013	3.0566
16	5.1	0.482	0.118	0.013	3.0588

17	5.162	0.511	0.12	0.013	3.017
20	5.127	0.496	0.131	0.013	3.0529
21	5.095	0.504	0.126	0.013	3.0815
23	5.107	0.524	0.121	0.013	3.072
24	5.166	0.523	0.133	0.013	3.0385
27	5.077	0.569	0.15	0.013	3.1139
32	5.095	0.573	0.142	0.013	3.0745

Table 2 shows the ferret program's behaviour for different thread counts. There is no significant change in context switching rate, CPU migration rate, or page faults observed during its execution. As a result, these parameters do not affect the program's execution time. The execution time is increased rapidly up to 12 threads, after which no changes are observed. This also implies that if all other parameters are constant, the programmer should select the number of threads with the highest CPU utilisation. The CPU utilisation is at its peak at 14 threads. This proves that, while the ferret program can be executed with 12 threads without loss, it is preferable to have 14 threads for better performance.

Table 3 shows the behaviour of swaptions. The CPU migration, context switching and page fault rates do not change significantly during execution. CPU utilisation is increased at thread count 16 and reached a peak at 23. We can see that the execution time increased at thread count 16, but not as much as it did at thread count 23. As a result, swaptions should be run with 16 threads because CPU utilisation is higher and execution time is not excessively increased.

Table 3. Effect if thread count on OS-level factors for swaptions program.

Thread count	CU	CW (M/sec)	CM (K/sec)	PF (M/sec)	ET (s)
6	2.403	0.402	0.101	0.027	1.8051
7	2.495	0.385	0.101	0.027	1.7177
10	2.868	0.51	0.095	0.025	1.6377
12	2.828	0.411	0.102	0.026	1.5812
13	2.848	0.003	0.106	0.026	1.5749
16	3.271	0.025	0.1	0.021	1.6588
17	3.222	0.018	0.094	0.022	1.6457
20	3.39	0.042	0.109	0.02	1.698
21	3.263	0.028	0.108	0.021	1.6513
23	3.759	0.067	0.112	0.017	1.7652
24	3.327	0.039	0.121	0.021	1.6682
27	3.525	0.065	0.136	0.019	1.7333
32	3.3	0.046	0.131	0.02	1.727

The programmer can set conditions and threshold values for OS-level factors by observing executions of streamcluster, ferret and swaption benchmarks. These conditions can be implemented in the fitness function, which tells whether or not the given thread count is a better solution.

The following steps are performed to obtain fitness values for manta rays

1. Get the multithreaded application and input data for which the thread count is to be determined.
2. Run the application with the same number of threads as indicated by manta rays's present position.
3. Set num_cores = number of cores, X_t = current position of a manta ray.
4. Collect the values for X_{cu}, X_{cs}, X_{cm}, X_{pf} and X_{et}.
5. $I_{pf} = X_{pf} - X_{best\text{-}pf}$; $I_{cu} = X_{cu} - X_{best\text{-}cu}$; $I_{et} = X_{et} - X_{best\text{-}et}$.
6. If ($X_{cs} <= N_0$) and ($X_{cm} <= N_1$) then return 0.
7. Else if ($I_{pf} > N_2 * X_{pf}$) then return 0.
8. Else if ($X_t >$ num_cores) and ($I_{cu} > 0$) and ($X_t < X_{best}$) and ($X_{et} < X_{best\text{-}et}$) then $X_{best} = X_t$; return 1.
9. Else if ($I_{cu} > N_3$ * num_cores)) and ($I_{et} < N_4 * X_{et}$)$ then $X_{best}=X_t$; return 1.
10. Return 0.

If the fitness function finds a new X_{best} it returns 1 else it returns 0. X_{cu}, X_{cs}, X_{cm}, X_{pf} and X_{et} indicate the CPU utilisation, context switching rate, CPU migration rate, page fault rate and execution time respectively. Programmer need to set values for N_0, N_1, N_2 and N_3 after observing the executions of sample programs. The lower threshold for CS is n_0, and the lower threshold for CM is N_1. If CU and CM are less than the thresholds, the function returns 0. After that, it checks for any drastic changes in PF, which means that if it is greater than ($N_2 * X_{pf}$), it returns 0. If the current thread count is less than X_{best} and CU is greater than $X_{best\text{-}cu}$ and ET is less than $X_{best\text{-}et}$, the current thread count becomes new X_{best} and the function returns 1. Finally, if CU is greater than $X_{best\text{-}cu}$ by N_3 times num_cores and no other factors are affected and all are under the thresholds, it returns 1.

5. Results and Discussions

Table 4. Experimental setup.

Server	Dual Socket Server
Processor	Intel Xeon E5 2603 v3
Number of Cores	12
Primary Memory	42GB
Operating System	Linux

The proposed method was evaluated on an Intel Xeon E5-2603 v3 server, a 12 cores system. Table 4 shows the experimental setup used in this study. Blackscholes, ferret, radiosity, swaptions, water_nsquared, water_spatial, x264, bodytrack, canneal, freqmine, raytrace, fmm, lu_cb, streamcluster and vips are among the 15 PARSEC benchmark programs tested using RMRFO. We selected a Linux-based system for our research because it provides a variety of tools for analysing application behaviour, such as perf. After determining the optimal thread count, the programs are tested with 'native' type input data, which is a large dataset available in the group. The algorithm's recommended thread count is found to be correct since all programs took less time to execute.

Table 5. Values for various variables and calculations done by RMRFO for water_spatial program

I	TY	R1	R2	R3	ID	PO	F
0	IN					12	1
1	CC	0.707			1	27	1
2	CC	0.17			2	16	0
3	CC	0.066			3	17	0
4	CC	0.902			4	12	0
5	SS		0.731	0.73	1	5	0
6	SS		0.241	0.258	2	14	0
7	SS		0.637	0.059	3	24	1
8	SS		0.55	0.244	4	20	1
9	CH	0.837			1	23	0
10	CH	0.063			2	10	0
11	CH	0.083			3	9	0
12	CH	0.392			4	17	0
13	SS		0.598	0.819	1	11	0
14	SS		0.333	0.371	2	12	0
15	SS		0.385	0.856	3	12	0
16	SS		0.985	0.715	4	15	0

The results obtained in 16 iterations for water_spatial benchmark is shown in Table 5 and Table 4. Table 5 shows the calculations done by RMRFO while searching for an optimal solution. CH, CC and SS indicate the three phases of algorithm: chain, cyclone and somersault respectively. The initial best position Xbest is set to an integer number equal to the number of cores in the system, which in this experiment is 12. The "I" column shows the iteration number and the "TY" column shows the type of foraging technique used by manta rays. The random numbers "R1" is used by cyclone and chain foraging while the random numbers "R2" and "R3" are used by somersault foraging. The columns "ID" and "PO" indicate the manta rays's identification number, number of threads and execution time respectively. The "F" column indicates whether the current manta ray is fit for the optimal solution. A value of 1 in this column indicates that a new solution is found.

Table 6. Values obtained by fitness functions for OS-level factors for different number of threads

I	N	CU	CW (M/sec)	CM (K/sec)	PF (M/sec)	ET (s)
0	12	3.714	0.258	0.046	0.012	2.667
1	27	3.942	0.306	0.065	0.012	2.537
2	16	3.916	0.265	0.054	0.012	2.533
3	17	3.744	0.269	0.053	0.012	2.686
4	12	3.714	0.258	0.046	0.012	2.667
5	5	2.663	0.235	0.047	0.013	3.627
6	14	3.61	0.264	0.049	0.012	2.76
7	24	3.943	0.298	0.062	0.012	2.536
8	20	3.953	0.279	0.058	0.012	2.523
9	23	3.629	0.283	0.056	0.012	2.778
10	10	3.563	0.251	0.045	0.012	2.776
11	9	3.198	0.25	0.047	0.013	3.053
12	17	3.744	0.269	0.053	0.012	2.686
13	11	3.267	0.254	0.045	0.012	3.058
14	12	3.71	0.258	0.046	0.012	2.667
15	12	3.71	0.258	0.046	0.012	2.667

The results obtained in 16 iterations for water_spatial benchmark is shown in Table 5 and Table 4. Table 5 shows the calculations done by RMRFO while searching for an optimal solution. CH, CC and SS indicate the three phases of algorithm: chain, cyclone and somersault respectively. The initial best position X_{best} is set to an integer number equal to the number of cores in the system, which in this experiment is 12. The "I" column shows the iteration number and the "TY" column shows the type of foraging technique used by manta rays. The random numbers "R1" is used by cyclone and chain foraging while the random numbers "R2" and "R3" are used by somersault foraging. The columns "ID" and "PO" indicate the manta rays's identification number, number of threads and execution time respectively. The "F" column indicates whether the current manta ray is fit for the optimal solution. A value of 1 in this column indicates that a new solution is found.

Table 6 shows calculations done by the fitness function for the same program. The X_{best} is initially set to 12 with a CU of 3.714. The X_{best} is changed to 27 in the next iteration because it has a higher CU value than the current $X_{best\text{-}cu}$. The algorithm discovered a better solution with 24 threads in the seventh iteration. In this case, the reason for selecting the said thread count is the higher CU value. In the following iteration, number 20 is chosen as X_{best} because it has a better CU and smaller ET than the corresponding values of the current X_{best} and all other parameters are almost unchanged. Finally, the algorithm returns the number 20 as the optimal solution which is much greater than the number of processors.

The performance of a parallel program is measured in terms of speedup. If a sequential program on a single core takes T(1) seconds to complete and a parallel version of the same program with N number of threads takes T(N) seconds, then speedup, S(N) is defined as

$$S(N) = T(1) / T(N) \qquad (7)$$

Table 7. Performance comparisons of PARSEC programs

Sr. No.	Benchmark Program	T(1)	T(12)	S(12)	N	T(N)	S(N)	I (%)
1	blackscholes (bs)	297.05	57.67	5.16	27	57.53	5.17	0.2
2	ferret (ft)	559.56	62.37	8.98	14	62.28	8.99	0.12
3	radiosity (rs)	300.34	272.1	1.11	17	266.6	1.13	1.81
4	swaptions (sw)	493.08	48.91	10.09	16	43.8	11.26	11.57
5	water nsquared (wn)	656.71	73.86	8.9	20	71.58	9.18	3.15
6	water spatial (ws)	268.5	38.15	7.04	20	36.36	7.39	4.98
7	x264 (x)	221.55	20.25	10.95	27	19.2	11.54	5.39

The speedup is calculated for all the benchmark programs to estimate the prediction accuracy of the proposed RMRFO prediction model. The speedup obtained with the new thread count is compared to the system's best thread count as shown in Table 7. Here N is the thread count obtained using RMRFO and considered an optimal solution. The improvement(I) in a speedup in RMRFO over the num_cores is defined as

$$I = (S(N) - S(num\_cores)) / S(num\_cores) \qquad (8)$$

where, num_cores = number of cores.

Seven of the fifteen programs tested from the benchmark had optimal thread counts greater than the number of processors in the system. Table 7 shows these programs and their speedup comparisons. We can see that swaptions have an optimal thread count of 23, with a more than 11.57 per cent improvement when compared to S(12). The water spatial and x264 programs both show a 5% improvement. Radiosity and water_nsquared have seen average improvements. There is no improvement in blackscholes and ferret, even though the algorithm suggested a thread count greater than 12, implying that it is safe to run these programs with the 27 and 14 threads.

Figure 4 shows the comparison of speedups between S(N) and S(12). The graph clearly shows that the speedup obtained from optimal thread count is greater than if the program was run with the number of threads equal to the number of cores. The graph in Figure 5 compares the number of iterations taken by the RMRFO and Thread-reinforcer to achieve the optimal thread count. The RMRFO has a fixed number of iterations, whereas the Thread-reinforcer starts with two threads and adds one thread in each iteration until it reaches the optimal count. The RMRFO in our study has taken 16 iterations. Therefore, if the number of processors is more than 16, the RMRFO will always perform better than Thread-reinforcer.

Table 8 lists the eight PARSEC programs for which the RMRFO has recommended a thread count equal to the number of processors. The freqmine has the maximum speedup of about 10, whereas raytrace and canneal have the lowest speedup of about 3. Except for streamcluster, all applications exhibit maximum CPU utilisation and minimal execution time at the optimal thread count. In the case of streamcluster, the context switching rate and CPU migration rate are below 0.001 for the number of threads more than 12.

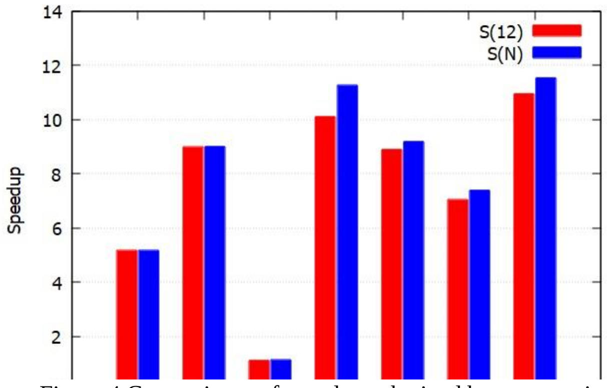

Figure 4 Comparisons of speedups obtained between optimal thread count and threads equal to number of cores.

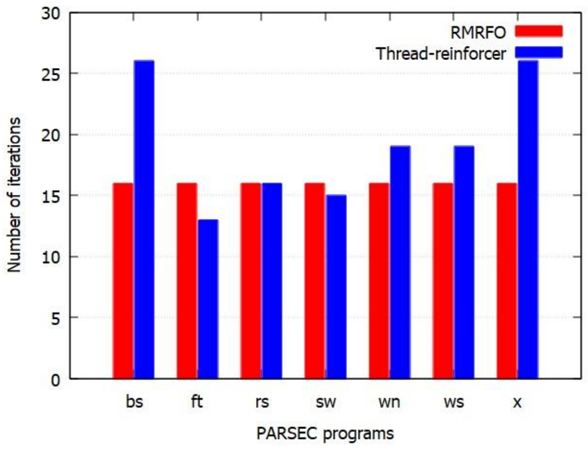

Figure 1. Comparisons of number of iterations performed by the RMRFO and Thread-reinforcer to obtain optimal thread count.

Table 8. list of the eight PARSEC programs for which the RMRFO has recommended a thread count equal to the number of processors and their performance.

Sr. No.	Benchmark	T(1)	N	T(N)	S(N)
1	bodytrack	259.04	12	49.37	5.25
2	canneal	335.63	12	116.31	2.89
3	freqmine	827.78	12	82.96	9.98
4	raytrace	412.79	12	150.08	2.76
5	Fmm	248.64	12	66.28	3.76
6	lu cb	242.12	12	27.42	8.84
7	streamcluster	1030.85	12	128.36	8.04
8	Vips	190.99	12	23.5	8.13

6. Conclusion

The simulation results show that the proposed RMRFO model explores the solution space and finds optimal solutions efficiently compared to the original Thread-reinforcer algorithm. In this study, the fitness function uses CPU utilisation, context switching rate, CPU migration rate, page fault rate and execution time to determine fitness value. The user must first run the programs with a small amount of data before running them with the actual input data to collect values of these factors. The fitness function uses streamcluser, ferret, and swaptions programs to determine various conditions and thresholds for the target hardware. We also discovered that 7 of the 15 programs examined have a higher optimal thread count than the number of processors. This method can be extended in the future by utilising deep learning strategies to analyse OS-level factors in real-time.

References

[1]. Navarro, Cristobal, et al. A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures. Communications in Computational Physics; 2013.

[2]. V.A. Chouliaras,T.R. Jacobs, J.L. Nu´n˜ez-Yanez, K. Manolopoulos, K. Nakos, D. Reisis. Thread-Parallel MPEG-2 and MPEG-4 Encoders for Shared-Memory System-On-Chip Multiprocessors. International Journal of Computers and Applications: Taylor & Francis; 2007. vol. 29. no. 4. p. 353–361.

[3]. S H Malave. Squid-SMP: Design & implementation of squid proxy server for the parallel platform. International Conference on Information Communication and Embedded Systems (ICICES2014); 2014. p. 1–6

[4]. Sajib Barua, Ruppa K. Thulasiram, Parimala Thulasiraman. High- Performance Computing for a Financial Application Using Fast Fourier Transform. Quality Technology & Quantitative Management: Taylor & Francis; 2014. vol 11. no, 1. p. 185–202

[5]. Ching-Kuang Shene. Multithreaded Programming Can Strengthen an Operating Systems Course. Computer Science Education: Routledge; 2002. vo. 12. no. 4. p. 275-299

[6]. Sethuraman S. Analysis of Fork-Join Systems. Network of Queues with Precedence Constraints (1st ed.) CRC Press; 2022.

[7]. Bhabani Shankar, Prasad Mishra, Satchidananda Dehuri. Parallel Com- puting Environments: A Review. IETE Technical Review: Taylor & Francis; 2011. vol. 28 no. 43 p. 240–247.

[8]. Mattson Tim. An introduction to openMP. Conference: Cluster Com- puting and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium; 2001.

[9]. Randell, Brian. Operating Systems: The Problems Of Performance and Reliability. 1971. p. 281–290.

[10]. Torsten Hoefler,Timo Schneider,Andrew Lumsdaine. Accurately mea- suring overhead, communication time and progression of blocking and nonblocking collective operations at a massive scale. International Journal of Parallel, Emergent and Distributed Systems: Taylor & Francis; 2010. vol. 24. no. 4. p. 241–258.

[11]. M. Dorigo, M. Birattari, T. Stutzle, Ant colony optimization. in IEEE Computational Intelligence Magazine; 2006. vol. 1. no. 4. p. 28–39.

[12]. Xin-She Yang, Particle Swarm Optimization. in IEEE Computational Intelligence Magazine. Academic Press; 2021. chapter 8. p. 111–121.

[13]. Weiguo Zhao, Zhenxing Zhang, Liying Wang. Manta ray foraging optimization: An effective bio-inspired optimizer for engineering appli- cations. Engineering Applications of Artificial Intelligence: 2020. vol. 87.

[14]. Min-Yuan Cheng, Doddy Prayogo. Symbiotic Organisms Search: A new metaheuristic optimization algorithm. Computers & Structures; 2014. vol 139. p. 98–112.

[15]. Xian-Bing Meng, X.Z. Gao, et al. A new bio-inspired optimisation al- gorithm: Bird Swarm Algorithm. Journal of Experimental & Theoretical Artificial Intelligence: Taylor & Francis; 2016. vol. 28 no. 4. p. 673–687.

[16]. Sukhpal Singh Gill, Rajkumar Buyya. Bio-Inspired Algorithms for Big Data Analytics: A Survey, Taxonomy, and Open Challenges. Big Data Analytics for Intelligent Healthcare Management, Academic Press; 2019. p. 1–17

[17]. C. Bienia, S. Kumar, J. P. Singh and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. 2008 Interna- tional Conference on Parallel Architectures and Compilation Techniques (PACT): IEEE; 2008. p. 72–81.

[18]. Pusukuri Kishore Kumar, Gupta Rajiv, Bhuyan Laxmi N. Thread Reinforcer: Dynamically Determining Number of Threads via OS Level Monitoring. IEEE Computer Society: 2011.

[19]. Qin Henry, et al. Arachne: Core-aware thread management. 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). 2018.

[20]. S. De Pestel, S. Van den Steen, S. Akram and L

Eeckhout. RPPM: Rapid Performance Prediction of Multithreaded Workloads on Multicore Processors. 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS); 2019, p. 257–267.

[21]. Xin Wei, Liang Ma, Huizhen Zhang, Yong Liu. Multi-core multi- thread-based optimization algorithm for large-scale travelling salesman problem. Alexandria Engineering Journal; 2021. vol. 60. no 1. p. 189–197,

[22]. R. Nath, D. Tullsen. Accurately modelling GPGPU frequency scaling with the CRISP performance model. In Emerging Trends in Computer Science and Applied Computing, Advances in GPU Research and Prac- tice Morgan Kaufmann. 2017. chapter 18. p. 471-505.[22] R. Nath, D. Tullsen. Accurately modelling GPGPU frequency scaling with the CRISP performance model. In Emerging Trends in Computer Science and Applied Computing, Advances in GPU Research and Prac- tice Morgan Kaufmann. 2017. chapter 18. p. 471-505.