International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 10 Issue: 12
DOI: https://doi.org/10.17762/ijritcc.v10i12.5889
***Article Received**: 16 October 2022 **Revised**: 05 November 2022 **Accepted**: 16 December 2022*

Symbiotic Organisms Search Optimization to Predict Optimal Thread Count for Multi-threaded Applications

Sachin H Malave[1], Subhash K Shinde[2]
[1]Reseach Scholar, Mumbai University
Lokmanya Tilak College of Engineering
New Mumbai, India
shmalave@apsit.edu.in
2Professor, Mumbai University
Lokmanya Tilak College of Engineering
New Mumbai, India
skshinde@ltce.in

Abstract— Multicore systems have emerged as a cost-effective option for the growing demands for high-performance, low-energy computing. Thread management has long been a source of concern for developers, as overheads associated with it reduce the overall throughput of the multicore processor systems. One of the most complex problems with multicore processors is determining the optimal number of threads for the execution of multithreaded programs. To address this issue, this paper proposes a novel solution based on a modified symbiotic organism search (MSOS) algorithm which is a bio-inspired algorithm used for optimization in various engineering domains. This technique uses mutualism, commensalism and parasitism behaviours seen in organisms for searching the optimal solutions in the available search space. The algorithm is simulated on the NVIDIA DGX Intel-Xeon E5-2698-v4 server with PARSEC 3.0 benchmark suit. The results show that keeping the thread count equal to the number of processors available in the system is not necessarily the best strategy to get maximum speedup when running multithreaded programs. It was also observed that when programs are run with the optimal thread count, the execution time is substantially decreased, resulting in energy savings due to the use of fewer processors than are available in the system..

Keywords- optimization, threads, parallel programming, benchmarks.

I. INTRODUCTION

Many OS and hardware factors have been observed to affect the thread performance, resulting in a direct detrimental impact on computer system throughput. In order to complete tasks efficiently in high-performance computing, parallel programs must utilize all of the cores available on the machine. Many times, the programmer's job is to select the optimal number of threads before starting the execution on the target machine. As a result, programmers devote a significant amount of time to investigating parallel program issues and determining the count of threads. Thus, rather than spending time analysing the programs for the target machine, programmers can use optimization algorithms based on bio-inspired optimization techniques to quickly find optimal thread count [1]. The bio-inspired algorithms are designed to identify the best answer in a known search space by mimicking the nature of organisms. These algorithms travel to various locations and attempt to arrive at the required points as quickly as possible.

Multithreading is a parallel programming technique used on a shared memory-multicore processor system. Programmers find parallelism in the serial code or rewrite the code to separate out the multiple sub-tasks to execute them parallelly on multiple processors. Parallel execution requires that work be partitioned [2], but partitioning requires great care. Also, dividing the multithreaded applications unevenly can lead to a single-threaded execution once the other running short threads have finished their execution. A parallel program may need to carry out synchronization in order to safely proceed with such the processing [3]. After dividing the program into small tasks, these tasks need to communicate with each other: after all, if a given thread did not communicate at all, it would have no effect and would thus not need to be executed [4]. However, because communication involves overhead, poor partition boundary selection can result in significant performance deterioration. Because each simultaneous thread consumes shared resources, such as space in processor caches, the number of simultaneous threads must frequently be limited. The processor caches will overload if too many threads run simultaneously, resulting in a higher cache miss probability, which will decrease performance [5]. On the other hand, when programmers need more numbers of threads, they have to overlap thread execution and I/O operations. Also, permitting

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 10 Issue: 12
DOI: https://doi.org/10.17762/ijritcc.v10i12.5889
***Article Received**: 16 October 2022 **Revised**: 05 November 2022 **Accepted**: 16 December 2022*

threads to execute concurrently greatly increases the program's complexity [6], which can make the program difficult to understand, degrading productivity.

Many difficulties in parallel programming have been observed by the researchers [7][8][9] in the last few years & many attempts have been made to provide solutions for these problems. It's difficult to write parallel programs without a strong understanding of parallel programming techniques. The researchers have developed many software tools to help programmers correctly identify the problems in parallelism [10][11]. The taxonomy for parallel processing performance problems in multi-core systems is presented in [12]. The problems can be classified into seven categories as Task Granularity, Synchronization, Data Sharing, Load Balancing, Data Locality, Resource Sharing and Input/Output. Task granularity refers to the number of threads within a task [13]. In parallel programs, it is challenging to find parallelism to fully utilize the capacity of the machine. Under task granularity [14]. oversubscription is one of the issues which occurs when the work of the application is divided into smaller tasks than required to exploit the multicore or multiprocessor platforms.

When a machine synchronizes threads that don't do enough work to justify the synchronization overhead, low work to synchronization ratio issue arises [15][16]. The lock contention and badly behaved spinlocks are the problems that arise in synchronization. When a thread tries to acquire a lock that is already acquired by some other thread, lock contention happens [17][18]. In most cases where a thread attempts to acquire a contended lock, the thread must wait for the lock to be released before the thread can continue execution. Thus, when locks contend, threads are blocked from executing until the lock becomes free. When the thread is already locked by spinlock, all other running threads that try to gain the same lock go into a loop waiting for the lock to become free, which can result in useless spinning around the loop [19][20].

II. LITERATURE REVIEW

AbdurRouf, et al [21] analyse the allocation of multiple threads on multiple processors. The Open MP style parallel programming API is being used to launch different numbers of threads. The performance is checked in single as well as multithreaded applications. Through various experiments on different multicore architectures, they found that the execution time of the program is reduced when the number of threads increased proportionally. Similarly, their studies suggest that when thread allocation is done correctly, performance improves. The thread count should be determined by the number of cores in the system, and it should be kept to be equal to the number of cores available.

Lim, et al [22] proposed a thread evolution kit (TEK) with a CPU mediator, stack tuner, and thread identifier for CE/IoT devices. Using this kit software developers can identify and correct the problems in parallel programs. When the program codes are compiled, the enhanced Thread Identifier inspects this information in order to properly manage each thread. This kit was checked on a CE/IoT development platform and compared to other approaches. However, because the APIs are also not POSIX compatible, they are challenging to convert to new CE/IoT systems.

Sethia et al [23] proposed an equaliser, which is a low-overhead device execution system that tracks an application's resource specifications and adjusts the on-chip parallel processing, processor speed, and memory bandwidth to meet the operating demands. It can save energy without sacrificing efficiency by throttling underutilized resources. It can also increase bottleneck resources to minimize contention and improve efficiency without significantly increasing energy consumption. Moreover, the performance of the proposed equalizer is very close to the DynCTA.

Qin et al [24]. has presented a solution that delivers both fast response time and throughput for programs with short threads. Based on system's load, each program decides how many cores it requires. It always knows precisely the cores it has been assigned, and it has complete control over the location of its threads on those cores. A central core arbiter manages the allocation of cores amongst programs.

Awatramani, et al [25] proposed a thread block scheduler to perform kernel-to-core mapping as well as scheduling thread blocks from active kernels on the mapped cores. When a new kernel starts or a kernel stops, the interleaved scheduler conducts an occupancy inspection. The maximum block area of all kernels within each centre is modified based on the results obtained.

Pusukuri, et al [26] developed a simple method called thread reinforcer for proactively calculating the required number of threads without resetting the program or changing Operating System rules. Since calculating the proper number of threads for a multithreaded program periodically is a difficult task. Furthermore, architectural specifications such as memory management issues are not considered here.

Sasaki et al. [27] created a complex scheduler that uses hardware monitoring and evaluation units to dynamically forecast application scalability and decides the best CPU cores to allot for each program. Because each program has its own set of characteristics different applications need different shared resources like processor cores and memory systems. As a result, it's apparent that OS thread scheduling becomes critical in attaining high systems throughput.

Heirman, et al [28] extended CRUST (Cluster-aware Undersubscribed Scheduling of Threads), a technique for determining the best thread count for OpenMP applications running on clustered cache architectures, for the Xeon Phi's

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 10 Issue: 12
DOI: https://doi.org/10.17762/ijritcc.v10i12.5889
***Article Received**: 16 October 2022 **Revised**: 05 November 2022 **Accepted**: 16 December 2022*

processor. By leveraging application phase behaviour at OpenMP parallel section borders, CRUST can automatically identify the optimal thread count at sub-application granularity. It also uses hardware performance counter information to get insight into the program's behaviour. This method controls the threads inside the CPU hardware cores at the moment of execution.

Kanemitsu, et al [29] proposed a task scheduling approach based on clustering for minimising the scheduled time in a large number of distinct processors. The number of processors utilized for actual execution is controlled in order to reduce the Worst Schedule Length (WSL). Real-time task assignment and task clustering are used to decrease the scheduling time until the total processing time in a task cluster reaches the lower limit. It concentrates on the near-optimal selection of processors without taking into account the impact of the number of active threads on system performance.

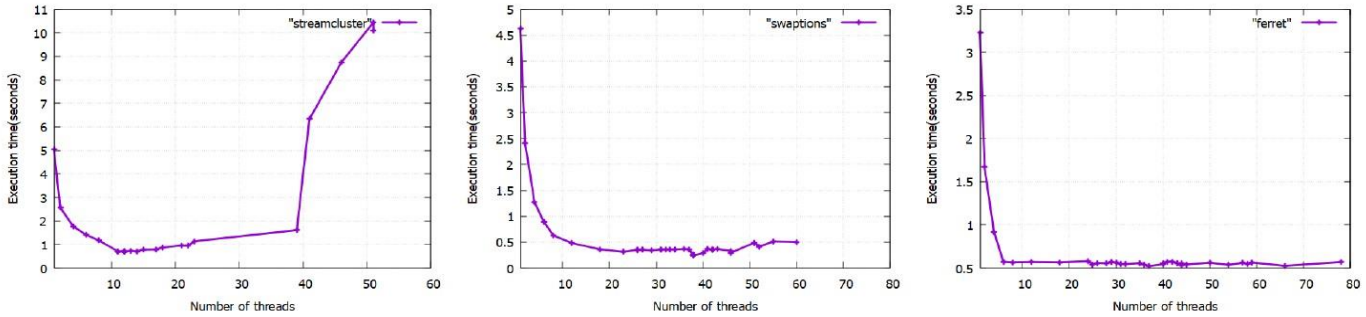

Figure 1

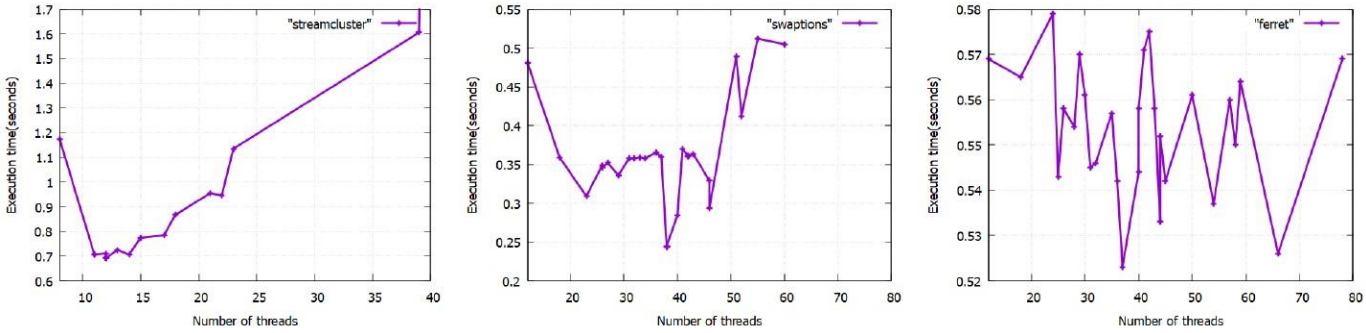

Figure 2

III. MOTIVATION AND PROBLEM STATEMENT

Figure 1 shows the graphs of execution time vs the number of threads on a 40 core Xeon processor system for streamcluster (left), swaptions (middle), and ferret (right) of the PARSEC benchmark. The X-axis represents the number of threads, while the Y-axis represents the execution time. The execution time lowers as the number of threads increases. We can see that the execution time for a certain number of threads reduced significantly at first but then fluctuated by a very random amount. This occurred because of the number of OS-level factors and their overheads interfering with and affecting the program's execution. It can be said that the streamcluster works well up to ten threads, but as the number of threads exceeds forty, it becomes unpredictable. Both swaptions and ferret operate excellently up to twenty and twelve threads respectively and do not greatly improve after that.

The areas in Figure 1 where these benchmarks ceased to progress as the number of threads grew are depicted in Figure 2. In order to get the best performance on multicore processing systems, the number of threads in multicore processing applications should always equal the number of cores. However, this is not the case with the benchmark results presented here. Because the execution time varies randomly as shown in Figure 2, no method or strategy can be used to find the thread count to minimize execution time. Thus, in this paper novel, searching optimization strategy for estimating an optimum number of threads in multithreaded programs is proposed to overcome this issue.

IV. PROPOSED METHODOLOGY

In [1], a population-based bio-inspired algorithm called Symbiotic Organism Search (SOS) was developed for addressing numerical optimization algorithms in the available search space. An SOS is a simulation of three symbiotic relationships between organisms: mutualism, commensalism, and parasitism. Mutualism refers to a relationship between two species in which both of them benefit from the

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 10 Issue: 12
DOI: https://doi.org/10.17762/ijritcc.v10i12.5889
***Article Received**: 16 October 2022 **Revised**: 05 November 2022 **Accepted**: 16 December 2022*

interaction. In a commensalism relationship, one organism benefits and another is not affected. In a parasitism relationship, one organism gets the benefits while the other one is harmed intentionally. The possible solutions represent the organism's positions in the search space, and these organisms always travel through the three phases described in SOS to reach new places. The fitness of these new positions is examined, and the best among all is selected as the better alternative. The user needs a fitness function that can assess the present position and calculates how near the organism is to reach the best solution. The user can choose a specific number of iterations or set certain exit criteria to get the desired results.

A. Phases in symbiotic organism search algorithm

In order to generate new solutions, the SOS optimization algorithm goes through three phases: mutualism, commensalism, and parasitism. The mathematical models for these phases are explained in this section.

1) Initialisation

The X_{best} is initialised to the number of cores available in the system in this modified SOS (MSOS) method, as it is typically considered that the number of threads should match the number of processors available in multicore systems. The search space is also constrained to the (Min, Max) pair. Here Min & Max represent minimum & maximum values allowed for organisms. The Max is set to two times the number of processors available in the target computer system. Eq. (1) is used to initialise the positions of organisms.

$$X(i) = r * (Max - Min) + Min \quad (1)$$

Where r is a random number between [0,1], X_{best} represents the optimal solution. The X is an Organism Vector and stores the current positions of the organisms inside the search space.

2) Mutualism phase

The interactions between humans and dogs are an example of mutualism, which directly benefits both organisms involved. The dogs are cared for and fed by humans and they guard humans against thievery and stranger attacks. This phase in the SOS algorithm mimics organism mutualism association. In this algorithm, X(i) is a number given to an organism that corresponds to the ecosystem's i^{th} position. The ecosystem's another organism, X(j), is chosen at random to associate with X(i). Both organisms have a mutualistic interaction in order to increase their mutual survival benefits in the system. The interaction between two organisms X(i) and X(j) is used to generate new alternative solutions called X(inew) and X(jnew). Following equations eq. (2) and eq. (3) define mathematical model for this phase.

$$X(inew) = X(i) + r * (X_{best} - (MV * BF1)) \quad (2)$$
$$X(jnew) = X(j) + r * (X_{best} - (MV * BF2)) \quad (3)$$

where,

$$MV = \frac{X(i) + X(j)}{2}$$

Here, r is a random number between range [0,1]. When organism X(i) interacts with organism X(j), it may gain a significant advantage from X(j). Meanwhile, while dealing with organism X(j), it may only receive adequate or minimal advantage. In this case, the values for both variables BF1 and BF2 are decided at random as 1 or 2. These variables describe the degree to which each creature benefits from the contact, and whether an organism gains partially or totally. For example, when organism X(i) interacts with organism X(j), it may gain a significant advantage. However, when organism X(j) interacts with organism X(i), it may only receive adequate or marginal benefit.

Here, MV is a Mutual Vector, and it reflects the association between the organisms X(i) and X(j). The Xbest represents position of organism where best fitness value was determined. As a result, we use Xbest to mimic the maximum degree of profitability as the objective point for both organisms. Finally, positions in Organism Vector are only updated if fitness values of new positions are better than the values before the interactions.

3) Commensalism phase

An arbitrary organism, X(j), is chosen from the search space to interact with X(i), analogous to the mutualism phase. In this case, organism X(i) tries to take advantage of the situation. In this phase the interaction, does not make any changes in other organism X(j). Following equations represents mathematical model for commensalism between two organisms.

$$X(inew) = X(i) + r * (X_{best} - X(j)) \quad (4)$$

Here, r is a random number and multiplies the benefit of X(j) over X(i) with respect to Xbest.

4) Parasitism phase

Here, X(inew) is formed in the solution space by replicating organism X(i) and then changing the randomly by allowing it to interact with a parasite organism X(j). Both X(i)and X(inew)have their fitness values examined, and if X(inew) has a better fitness than X(i), then X(inew) completely replaces X(i).

$$X(inew) = X(i) + r * X \quad (5)$$

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 10 Issue: 12
DOI: https://doi.org/10.17762/ijritcc.v10i12.5889
***Article Received**: 16 October 2022 **Revised**: 05 November 2022 **Accepted**: 16 December 2022*

B. Msos alogorithm

The steps in proposed algorithm to predict optimal threads count are explained below.

1. Set the count of organisms (N) to a positive number.
2. Use Eq. (1) to determine the locations of all organisms and put them into Organism Vector.
3. Set the initial best solution equal to the number of compute cores available. (Xbest)
4. Repeat until the maximum number of iterations have been completed or the desired answer has been obtained.
5. For each organism, i
6. Perform Mutualism using eq. (2) and eq. (3). If the newly determined positions have fitness values less than the prior positions, then update the Organism Vector with new positions.
7. Perform Commensalism using eq. (4). If the newly determined positions have fitness values less than the prior positions, then update the Organism Vector with new positions.
8. Perform parasitism using eq. (5 If the newly determined positions have fitness values less than the prior positions, then update the Organism Vector with new positions.
9. The optimal number of threads is determined by the best position found in the preceding steps.

Above algorithm show the working of MSOS, which includes all detailed information. All organisms are allowed to go through all three steps mentioned in MSOS to determine new positions, and the best position among all is chosen as the current best solution. This process is repeated until the maximum number of iterations have been performed or an exit condition has been fulfilled. The fitness value here refers to the time it takes for the program to run with the provided number of threads. To calculate the fitness value for organisms, following steps are performed.

1. Determine the Organism's current location.
2. Get the parallelized application and its sample input data to determine the thread count.
3. Run the program with the number of threads equal to the current position indicated by the organism.
4. The fitness value of an organism is the time it takes to complete the task.

When compared to other conventional techniques like machine learning (ML), the suggested MSOS-based thread predictions model has various advantages, including being modest, low overhead, and dynamic in forecasting thread count (SR). Because ML-based techniques require greater training time to achieve better prediction accuracy, they have the higher overheads.

The proposed method is capable of achieving the best combination of prediction accuracy and low overheads.

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 10 Issue: 12
DOI: https://doi.org/10.17762/ijritcc.v10i12.5889
Article Received: 16 October 2022 Revised: 05 November 2022 Accepted: 16 December 2022

Table 1 . Iteration wise positions and fitness calculations for streamcluster benchmark

Iteration: i	Phase	New Position of i^{th} Organism: Fitness	New Position of j^{th} Organism: Fitness	Organism Vector Index: [0 1 2 3 4]	Fitness Vector Index: [0 1 2 3 4]	X_{best}
0:0	M	41:06.4	38:1.526	[40,61,38,24,33]	[1.819,12.196,1.526,1.041,1.644]	24
	C	53:10.485		[40,61,38,24,33]	[1.819,12.196,1.526,1.041,1.644]	
	P	69:17.444		[40,61,38,24,33]	[1.819,12.196,1.526,1.041,1.644]	
0:1	M	43:07.0	23:0.917	[23,43,38,24,33]	[0.917,7.048,1.526,1.041,1.644]	23
	C	52:10.192		[23,43,38,24,33]	[0.917,7.048,1.526,1.041,1.644]	
	P	63:13.859		[23,43,38,24,33]	[0.917,7.048,1.526,1.041,1.644]	
0:2	M	31:01.2	17:0.809	[23,43,31,17,33]	[0.917,7.048,1.230,0.809,1.644]	17
	C	37:1.464		[23,43,31,17,33]	[0.917,7.048,1.230,0.809,1.644]	
	P	46:8.352		[23,43,31,17,33]	[0.917,7.048,1.230,0.809,1.644]	
0:3	M	17:00.8	17:0.812	[23,43,31,17,33]	[0.917,7.048,1.230,0.773,1.644]	
	C	18:0.771		[23,43,31,18,33]	[0.917,7.048,1.230,0.771,1.644]	18
	P	25:1.013		[23,43,31,18,33]	[0.917,7.048,1.230,0.771,1.644]	
0:4	M	31:01.2	15:0.815	[23,43,31,18,31]	[0.917,7.048,1.230,0.771,1.227]	
	C	41:6.325		[23,43,31,18,31]	[0.917,7.048,1.230,0.771,1.227]	
	P	51:9.968		[23,43,31,18,31]	[0.917,7.048,1.230,0.771,1.227]	
1:0	M	14:00.7	32:1.283	[14,32,31,18,31]	[0.694,1.283,1.230,0.771,1.227]	14
	C	10:0.777		[14,32,31,18,31]	[0.694,1.283,1.230,0.771,1.227]	
	P	16:0.891		[14,32,31,18,31]	[0.694,1.283,1.230,0.771,1.227]	
1:1	M	25:01.0	29:1.107	[14,25,31,18,31]	[0.694,1.014,1.230,0.771,1.227]	
	C	24:1.025		[14,25,31,18,31]	[0.694,1.014,1.230,0.771,1.227]	
	P	34:1.701		[14,25,31,18,31]	[0.694,1.014,1.230,0.771,1.227]	
1:2	M	26:01.3	11:0.766	[14,25,31,18,31]	[0.694,1.014,1.230,0.771,1.227]	
	C	41:6.171		[14,25,31,18,31]	[0.694,1.014,1.230,0.771,1.227]	
	P	35:1.806		[14,25,31,18,31]	[0.694,1.014,1.230,0.771,1.227]	
1:3	M	17:00.8	22:0.966	[14,25,22,17,31]	[0.694,1.014,0.966,0.756,1.227]	
	C	30:1.462		[14,25,22,17,31]	[0.694,1.014,0.966,0.756,1.227]	
	P	20:0.921		[14,20,22,17,31]	[0.694,0.921,0.966,0.756,1.227]	
1:4	M	29:01.4	19:0.834	[14,19,22,17,31]	[0.694,0.834,0.966,0.756,1.227]	
	C	33:1.643		[14,19,22,17,31]	[0.694,0.834,0.966,0.756,1.227]	
	P	58:11.377		[14,19,22,17,31]	[0.694,0.834,0.966,0.756,1.227]	

V. RESULTS AND DISCUSSION

Table 2 Experimental Setup

Server	NVIDIA DGX STATION
Number of Physical Cores	20
Number of Logical Cores	40
Main Memory	256 GB
Operating System	Linux

The proposed MSOS technique is tested on PARSEC, a set of well-known benchmark program. The ferret, freqmine, streamcluster, swaptions, vips, and vorland are among the six benchmarks taken from the PARSEC suite for illustration here. Our experimental setup, which comprises of an Intel Xeon-E5-2698-v4 2.2 GHz processor, is shown in Table 2. This computer has 40 logical cores and 256 GB of main memory. We chose the Linux based computer system for our study because it offers a wide variety of tools (ps, top, vmstat, etc) for understanding and analyzing application behaviors. Each experiment was repeated ten

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 10 Issue: 12
DOI: https://doi.org/10.17762/ijritcc.v10i12.5889
***Article Received**: 16 October 2022 **Revised**: 05 November 2022 **Accepted**: 16 December 2022*

times and the findings were averaged. For each program, the PARSEC benchmark specifies six input data sets. The "simlarge" input dataset is used in MSOS's fitness function to determine the execution time.

Table 1 shows the results obtained using MSOS up to two iterations. The Iteration column in this table shows the iteration numbers as well as the indices of organisms selected from Organism Vector. The types of interactions applied on organisms are listed in “Phase” column. Here, Mutualism, Commensalism, and Parasitism phases are represented by letters M, C and P respectively. The positions obtained after interacting with the other organisms are shown in column “New Position of ith Organism: Fitness”. This column also shows the fitness values of the new positions. The column "New Location of jth Organism: Fitness" indicates new positions determined for the randomly selected jth organisms from the Organism Vector. This is done during the Mutualism phase. The fitness values are also shown in the same column. The values in Organism Vector after applying the chosen interaction types are shown in “Organism Vector" column. In this example, Organism Vector contains five organisms. The "Fitness vector" column displays fitness values of organisms in Organism Vector. Finally, “Xbest" the last column displays the best solution found after applying MSOS phases.

Figure 3 shows graphically all the places visited by organisms during the execution of MSOS & Figure 4 depicts their fitness values. It is apparent that the places where the fitness values were close to the required solution (low in this case) were visited more frequently. This also demonstrates that MSOS tries to explore areas frequently where good solutions exist. The algorithm has jumped to high positions from time to time to avoid getting trapped in local minima. The MSOS examined the relevant space 36 times out of 45 visits, accounting for 80 per cent of all efforts.

In parallel programs, speedup is a great way to evaluate performance. If a sequential program on a single core takes T(1) seconds and a parallel program on N processors takes T(N) seconds, then Speedup, S(N), is defined as

$$S(N) = T(1)/T(N) \quad (6)$$

The speedup is calculated for the same benchmark programs to determine the prediction accuracy of the proposed MSOS based forecasting model. Table 3 shows the speedup gained after running the benchmark programs with the optimum & 40 number of threads. The column N shows optimum thread count determined by MSOS. The improvement in speedups obtained using MSOS and traditional method is defined as follows:

$$\delta = (S(N) - S(num\_cores))/S(num\_cores) \quad (7)$$

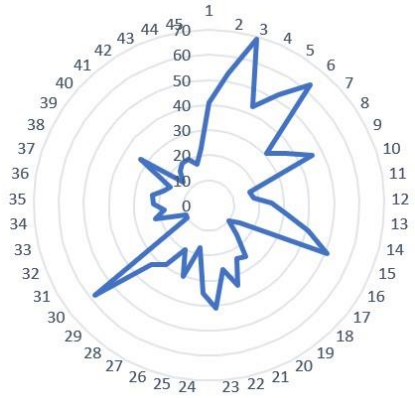

Figure 3 Streamcluster: Positions obtained for organisms

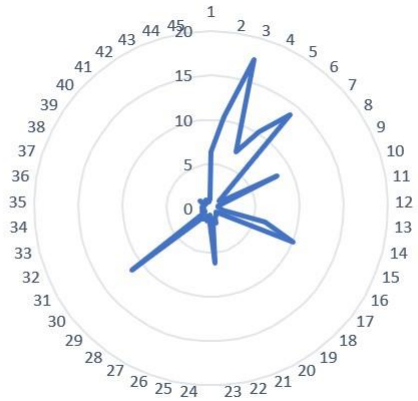

Figure 4 Streamcluster: Fitness values obtained for organisms

The results produced with MSOS are better compared to those obtained by keeping the thread count equal to the core count of the machine. The comparison between S(40) & S(N) is shown graphically in figure 5. The X-axis in this graph represents the names of benchmark programs while the Y-axis represents speedups. The ferret, streamcluster, freqmine, swaptions, and vips work much better when the thread count is less than 40. The vorland outperformed the others with a thread count of 57. When compared to running with 40 threads, streamcluster enhanced by 127 percent. The streamcluster, swaptions and vorland have all seen significant improvements in speedups. Figure 6 depicts graphically the comparison between the execution time required for optimal thread count & single thread (T(1) vs T(N)). The X-axis in this graph represents the names of benchmark programs while the Y-axis represents execution time in seconds. The vorland's execution time has been adjusted down to accommodate correctly in the graph. All of the benchmarks show an increase in performance.

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 10 Issue: 12
DOI: https://doi.org/10.17762/ijritcc.v10i12.5889
***Article Received**: 16 October 2022 **Revised**: 05 November 2022 **Accepted**: 16 December 2022*

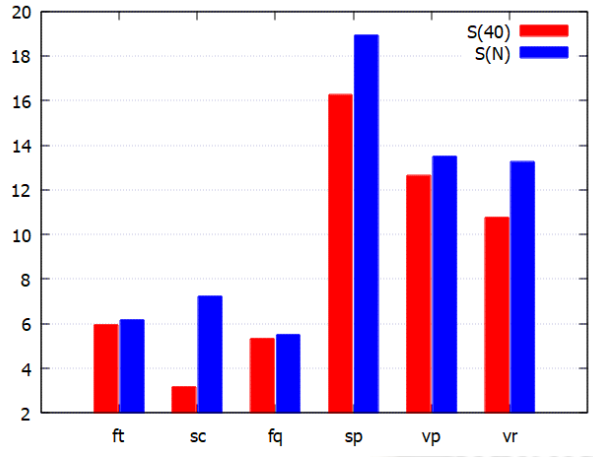

Figure 5 S(40) vs S(N)

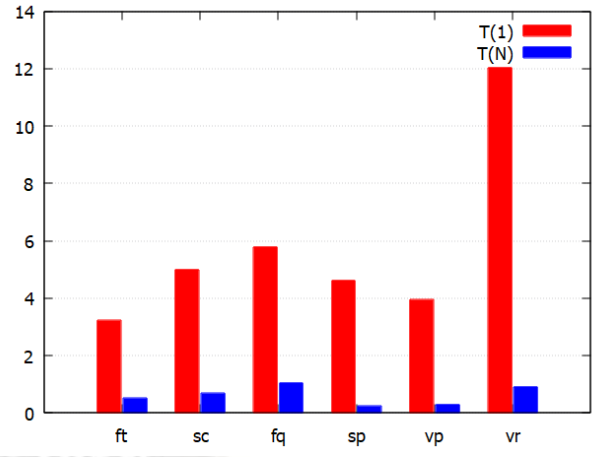

Figure 6 T(1) vs T(N)

Table 3 Performance analysis of PARSEC benchmarks

Benchmarks	**T(1)**	**T(40)**	**S(40)**	**N**	**T(N)**	**S(N)**	***δ***
Ferret(ft)	3.231	0.543	5.950	37	0.523	6.177	3.82%
Streamcluster(sc)	5.007	1.573	3.183	14	0.694	7.246	127.64%
Freqmine(fq)	5.785	1.084	5.336	28	1.05	5.509	3.24%
Swaptions(sp)	4.619	0.387	11.935	38	0.244	18.930	58.61%
Vips(vp)	3.946	0.312	12.647	38	0.292	13.513	6.85%
Vorland(vr)	120.44	11.192	10.761	57	9.08	13.265	23.26%

VI. Conclusion

In this paper, an efficient and unique threads prediction model is developed based on the MSOS algorithm. The prediction model determines the optimal number of threads for maximum speedups with ease. The simulation findings demonstrate that the proposed algorithm can efficiently find the available search space and swiftly converges to an optimal solution. The method described in this paper is straightforward, but it has some downsides as the user must run the program with a small quantity of data before running it with the actual input data. These overheads become minor if the actual input data is extremely large. Therefore, while using MSOS to evaluate an application, it's critical to choose the right amount of data.

References

[1] Min-Yuan Cheng, Doddy Prayogo. “Symbiotic Organisms Search: A new metaheuristic optimization algorithm.” Computers and Structures, vol. 139, pp 98-112, July 2014.

[2] Yan, Chenggang, et al. "A highly parallel framework for HEVC coding unit partitioning tree decision on many-core processors." IEEE Signal Processing Letters 21.5 (2014): 573-576.

[3] Lim, Amy W., and Monica S. Lam. "Maximizing parallelism and minimizing synchronization with affine transforms." Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 1997.

[4] Grant, Ryan E., et al. "Finepoints: Partitioned multithreaded mpi communication." International Conference on High Performance Computing. Springer, Cham, 2019.

[5] Martinez, Jose F., and Josep Torrellas. "Speculative synchronization: Applying thread-level speculation to explicitly parallel applications." ACM SIGOPS Operating Systems Review 36.5 (2002): 18-29.

[6] P. E. McKenney, M. Gupta, M. Michael, P. Howard, J. Triplett, and J. Walpole, “Is parallel programming hard, and if so, why?” Portland State University, Computer Science Department, Tech. Rep., TR-09-02, Feb. 2009.

[7] R. Atachiants, D. Gregg, K. Jarvis, and G. Doherty, “Design considerations for parallel performance tools,” in Proc. SIGCHI Conf. Human Factors Compute. Syst., 2014, pp. 2501–2510.

[8] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr., “Does bug prediction support human developers? Findings from a Google case study,” in Proc. Int. Conf. Softw. Eng., 2013, pp. 372–381.

[9] Ko and B. Myers, “An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks,” IEEE Trans. Softw. Eng., vol. 32, no. 12, pp. 971–987, Dec. 2006

[10] Anderson, Thomas E., and Edward D. Lazowska. "Quartz: A tool for tuning parallel program performance." ACM SIGMETRICS Performance Evaluation Review 18.1 (1990): 115-125.

[11] Navarro, Cristobal A., Nancy Hitschfeld-Kahler, and Luis Mateu. "A survey on parallel computing and its applications in data-parallel problems using GPU

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 10 Issue: 12
DOI: https://doi.org/10.17762/ijritcc.v10i12.5889
***Article Received**: 16 October 2022 **Revised**: 05 November 2022 **Accepted**: 16 December 2022*

architectures." Communications in Computational Physics 15.2 (2014): 285-329.

[12] Roman Atachiants, Gavin Doherty, and David Gregg., "Parallel performance problems on shared-memory multicore systems: taxonomy and observations," IEEE Trans. Softw. Eng., vol. 42, no. 8, pp. 764–785, Aug. 2016.

[13] Muthuvelu, Nithiapidary, et al. "On-line task granularity adaptation for dynamic grid applications." International Conference on Algorithms and Architectures for Parallel Processing. Springer, Berlin, Heidelberg, 2010.

[14] Iancu, Costin, et al. "Oversubscription on multicore processors." 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS). IEEE, 2010.

[15] Li, Jian, Jose F. Martinez, and Michael C. Huang. "The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors." 10th International Symposium on High Performance Computer Architecture (HPCA'04). IEEE, 2004.

[16] Sridharan, Srinivas, Arun Rodrigues, and Peter Kogge. "Evaluating synchronization techniques for light-weight multithreaded/ multicore architectures. " Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures. 2007.

[17] N. Tallent, J. Mellor-Crummey and A. Porterfield, "Analyzing lock contention in multithreaded applications," in Proc. 15th ACM SIGPLAN Symp. Principles Practice Parallel Program., 2010, pp. 269–280.

[18] Amer, Abdelhalim, et al. "Lock contention management in multithreaded mpi." ACM Transactions on Parallel Computing (TOPC) 5.3 (2019): 1-21.

[19] Cui, Yan, et al. "Lock-contention-aware scheduler: A scalable and energy-efficient method for addressing scalability collapse on multicore systems." ACM Transactions on Architecture and Code Optimization (TACO) 9.4 (2013): 1-25.

[20] Venugopal, Srikumar, Rajkumar Buyya, and Kotagiri Ramamohanarao. "A taxonomy of data grids for distributed data sharing, management, and processing." ACM Computing Surveys (CSUR) 38.1 (2006): 3-es.

[21] AbdurRouf, Mohammad, et al. "Performance Improvement using Optimal Thread Allocation Algorithm in Multicore Processor." (2018)

[22] Lim, Geunsik, Donghyun Kang, and Young Ik Eom. "Thread Evolution Kit for Optimizing Thread Operations on CE/IoT Devices." IEEE Transactions on Consumer Electronics 66.4 (2020): 289-298.

[23] Sethia, Ankit, and Scott Mahlke. "Equalizer: Dynamic tuning of gpu resources for efficient execution." 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE, 2014.

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

[25] Awatramani, Mihir, Joseph Zambreno, and Diane Rover. "Increasing gpu throughput using kernel interleaved thread block scheduling." 2013 IEEE 31st International Conference on Computer Design (ICCD). IEEE, 2013.

[26] Pusukuri, Kishore Kumar, Rajiv Gupta, and Laxmi N. Bhuyan. "Thread reinforcer: Dynamically determining number of threads via os level monitoring." 2011 IEEE International Symposium on Workload Characterization (IISWC). IEEE, 2011.

[27] Sasaki, Hiroshi, et al. "Scalability-based manycore partitioning." Proceedings of the 21st international conference on Parallel architectures and compilation techniques. 2012.

[28] Heirman, Wim, et al. "Automatic SMT threading for OpenMP applications on the Intel Xeon Phi co-processor." Proceedings of the 4th international workshop on runtime and operating systems for supercomputers. 2014.

[29] Kanemitsu, Hidehiro, Masaki Hanada, and Hidenori Nakazato. "Clustering-based task scheduling in a large number of heterogeneous processors." *IEEE Transactions on Parallel and Distributed Systems* 27.11 (2016): 3144-3157.

[30] Birhanu, Thomas Mezmur, et al. "Efficient thread mapping for heterogeneous multicore iot systems." *Mobile Information Systems* 2017 (2017).