Accelerating Global Optimization: A Parallelized approach to Particle Swarm Optimisation

Karunya Harikrishnan
B.TECH AI AND DS A
21011101059

Keerthana G
B.TECH AI AND DS A
21011101061

I. Problem Statement

PSO parallelization involves optimizing the Particle Swarm Optimization algorithm for efficient execution on parallel computing architectures. This entails devising strategies to distribute computation tasks across multiple processing units while maintaining communication and synchronization among them. The aim is to enhance PSO's scalability, speed and ability to handle large-scale optimization problems.

II. Particle Swarm Optimisation

The Serialised PSO algorithm can be summarized as follows:

1: Initialize population randomly
2: Evaluate cost function for each particle's position
3: Initialize personal best positions (pBest) for each particle
4: Initialize global best position (gBest) based on pBest values
5: **while** termination criteria not met **do**
6: **for** each particle **do**
7: Update velocity based on personal and global best positions
8: Update position based on velocity
9: Evaluate cost function for new position
10: **if** current position is better than personal best **then**
11: Update personal best position
12: **end if**
13: **if** current position is better than global best **then**
14: Update global best position
15: **end if**
16: **end for**
17: **end while**

III. Approach

In this project PPSO (Parallel PSO) is implemented using OpenMP and MPI frameworks using 2 different algorithms and the average execution time of ten independent Monte Carlo runs is compared as a performance indicator.

The 2 approaches we used are summarized as follows:

A. Approach 1

Message Passing Interface based parallel Computation and PSO optimisation algorithm are combined to form the PPSO.

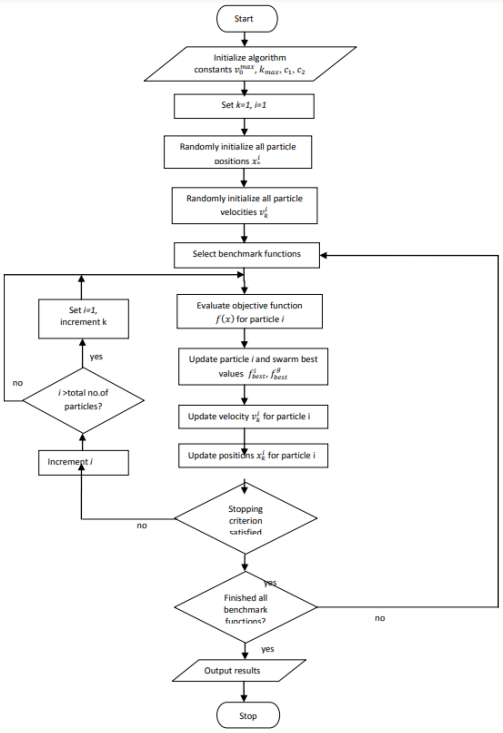

Fig. 1. Flowchart of Serial PSO

Here are the steps involved in MPI based parallel computation using processors:

- Processor 0 reads and pre-processes all the data before scattering them to other processors' memory.
- After processor 0's initialization work, all the processors calculate with the data that are sent by processor 0 through message passing interface.
- When the calculations are done, results are sent back to processor 0's receiving buffer to be dealt with. Then processor 0 decides whether to continue computing or output final results and terminate the program.

The optimization program flow can be described as follows (assuming there are n processors running this program simultaneously):

- Processor 0 is the master node. The others are slave nodes.

- Processor 0 and all slave nodes initialize PPSO parameters and MPI environment.
- Processor 0 randomly generates a swarm of particles, each of which stands for a candidate solution to a minimization or maximization problem. Let the size of the particle population be p.
- Let $m = \frac{p}{n}$. If n cannot be divided exactly by p, let $m = \frac{p}{n} + 1$, thus $n \times m = p$. Processor 0 partitions the swarm to n parts, each of which has m or $m-1$ particles. Then processor 0 sends these parts to other processors using `MPI_Scatter()` function. Each processor receives m or $m-1$ particles from processor 0 using the same `MPI_Scatter()` function.
- Each processor evaluates the fitness of its own particles based on the particular function.
- Processor 0 gathers the fitness values of all particles from processor 1 to processor $n-1$ using the `MPI_Gather` function.
- Processor 0 executes the PSO operator and adjusts all particles' positions in the search hyperspace.
- Processor 0 outputs the best solution of the current time interval. If the time interval counter is not equal to the maximum iteration (Nt), then set the counter to be $Nt+1$.
- Processor 0 sends the global best solution to all the other processes using the `MPI_Bcast()` function. This value is received by the other processes for the calculations in the further iterations.
- Go to step 3 if the stopping criteria are reached. Otherwise, go to step 10.
- All processors exit the MPI environment and terminate the program.

B. Approach 2

The OpenMP package contains compiler directives, run time routines and environment variables. Since all data is recorded on the globally shared memory on the CPU, OpenMP programming is much simpler.

- All the particles are initialized with the initial parameters using the `omp for` construct. Global Best value is calculated using the reduction clause.
- In each iteration, calculation for each particle is shared between different threads using the `omp for` construct. The number of threads to be used is predefined.
- Each local best is considered the global best position among them is calculated sequentially for proper synchronization.
- These steps are repeated until we reach the most optimal value or the maximum number of iterations, whichever is reached first.

IV. Performance metrics - Benchmark functions

Ackley's function has one narrow global optimum basin and many minor local optima. It is probably the easiest problem as its local optima are neither deep nor wide. In this project

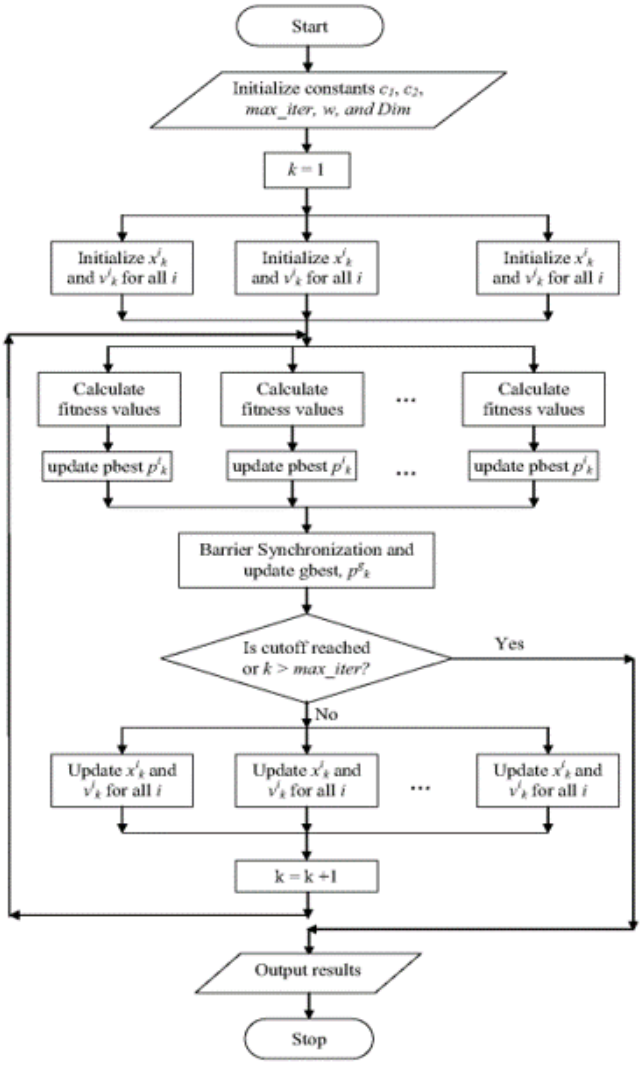

Fig. 2. Approach 1

we have used the Ackley Function as our benchmark function The equation of the function is as follows:

$$f(x,y) = -20\exp\left(-0.2\sqrt{x^2+y^2}\right) - \exp\left(0.5(\cos^2(x)+\cos^2(y))\right) + e + 20 \quad (1)$$

The function has a global minimum at f = 0 and x = (0, 0, 0, 0). It has one narrow optimum basin and many minor local optima.

V. Results

Two different types of results were calculated to see the parallelization effect in terms of

- Speed up based on Number of Iterations
- Speed up based on Number of Particles

The comparisons of the 2 approaches were made with the time taken for the original PSO Code run on a single core without parallelisation.

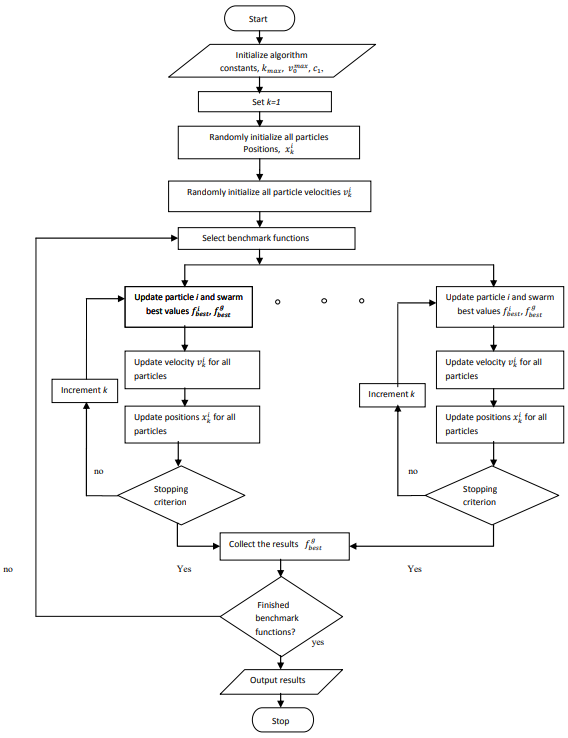

Fig. 3. Approach 2

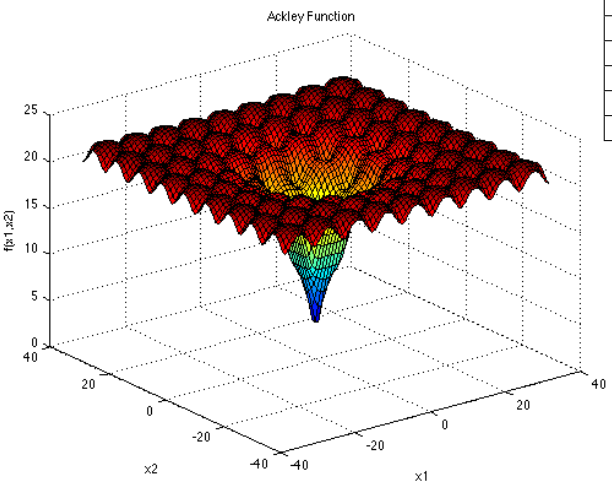

Fig. 4. Ackley Function

A. Speed up based on Number of iterations:

The experiment compares PSO implementations with a constant number of particles (=100) and dimensions (=5) but varying numbers of maximum iterations. The table shows the results of the experiment. The findings are that

- As the number of iterations increases, the parallel code achieves significant speedup upto 100 percent. This is because the parallel code has some overhead for a smaller number of iterations, but this overhead becomes negligible as the number of iterations increases.
- Both OpenMP and MPI implementations show reduced execution time as the number of threads or processes is increased, with the least time being observed at four threads or processes. This is because using more threads or processes increases the parallelization.
- MPI takes less time than OpenMP

B. Speed up based on number of particles

In the experiment, both serial and parallel PSO were run with a constant number of iterations (2000) and dimensions (10), but the number of maximum iterations was varied. The following inferences can be made

- MPI and OpenMp achieve more speedup up than the last experiment with MPI speeding upto three times the serial code. This is because the parallelisation is done primarily on particles.
- In both OpenMp and MPI, the time execution reduces as no of threads/processes are increased with least time at four threads/processes i.e. due to increase in parallelisation. Also there is drastic change between time execution of two threads/process and four threads/process,hence the number of parallelisation directly affects time execution.

Iterations	Time						Serial
	OpenMP			MPI			
	2	3	4	2	3	4	
500	0.0032	0.0029	0.036	0.031	0.03	0.036	0.044
1000	0.058	0.054	0.054	0.058	0.052	0.052	0.08
2000	0.11	0.1	0.1	0.095	0.09	0.085	0.0156
4000	0.205	0.196	0.165	0.175	0.17	0.145	0.303
8000	0.405	0.385	0.35	0.35	0.33	0.29	0.595

Particles	Time						Serial
	OpenMP			MPI			
	2	3	4	2	3	4	
500	1.15	0.9	0.73	0.81	0.75	0.65	1.5
1000	2.22	1.76	1.52	1.82	1.50	1.25	2.96
2000	4.54	3.65	3.01	3.17	3.08	2.76	6.31
4000	9.23	7.19	5.75	6.42	6.07	4.91	11.81
6000	13.86	11.22	10.1	9.67	9.53	7.35	17.82

VI. Conclusion

This suggests that parallel execution of the Particle Swarm Optimization (PSO) algorithm is most beneficial under specific conditions:

- High Number of Particles and Iterations: When the number of particles and iterations in the PSO algorithm is very

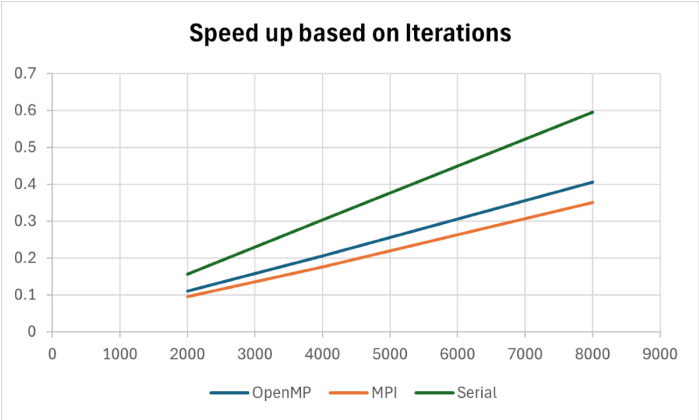

Fig. 5. Approach 1 Speed up Graph

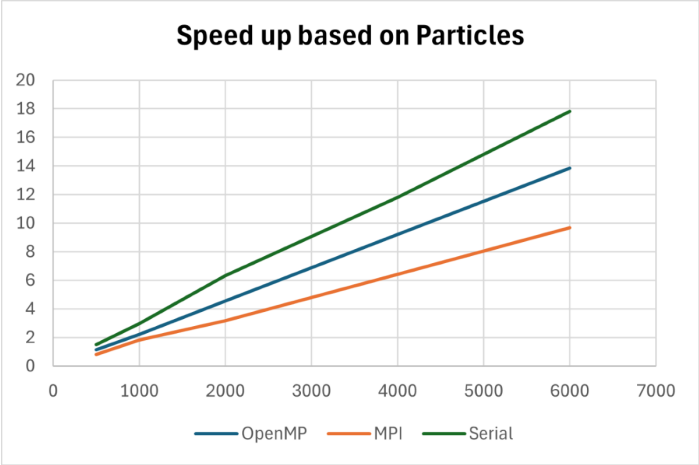

Fig. 6. Approach 2 Speed up Graph

high, parallel execution becomes more efficient. This is likely because with a high number of particles and iterations, there is a significant amount of computational work to be done, and parallelization allows for distributing this workload across multiple processing units, thus reducing the overall execution time.

- Impact on Number of Particles vs. Iterations: The statement suggests that parallelization has a greater effect on the performance of PSO in terms of the number of particles rather than the number of iterations. This could imply that increasing the number of particles being optimized simultaneously has a more noticeable impact on the efficiency gains achieved through parallel execution compared to increasing the number of iterations. In other words, parallelization scales better with the number of particles.

Overall, these findings emphasize the importance of considering both the number of particles and iterations when deciding whether to parallelize the PSO algorithm.

VII. References

1) https://library.ndsu.edu/ir/bitstream/handle/10365/25649/Parallel
2) https://doi.org/10.1155/2013/756719
3) https://www.catalyzex.com/paper/arxiv:2005.00863
4) https://www.sciencedirect.com/science/article/abs/pii/S15684946230
5) https://ieeexplore.ieee.org/document/9248594
6) https://github.com/jynxmagic/Parallel-Particle-Swarm-Optimization/tree/master