Parallel Computation of Particle Swam Optimization (PSO) using MPI

Project Report (CS-5234 Spring 2024)

Lalit Prasad Peri (lalitprasad@vt.edu)

1.Introduction to PSO

Particle Swarm Optimization (PSO) stands as a metaheuristic algorithm meticulously crafted to pinpoint the pinnacle of continuous nonlinear functions, drawing its inspiration from the intricate workings of biological systems and the collective intelligence observed in swarm behavior. Initially conceived by Kenney and Eberhart in 1990, the algorithm has since undergone a series of refinements and enhancements spearheaded by Eberhart, Kennedy, Shi, and other luminaries in the field. Its versatile nature finds application across diverse domains including but not limited to molecular biology, drone intelligence, and meteorology. Central to PSO is the concept of a swarm, where each constituent entity is referred to as a particle. These particles collectively contribute to the pursuit of the global optimization solution, referred to as the Global Optima. Moreover, particles engage in a perpetual process of self-evaluation, continuously refining their own local solutions, termed as Local Optima, to converge towards the overarching global solution. The project under discussion specifically implements asynchronous PSO methodology, a sophisticated approach geared towards computing the swarm's Local Optima within a two-dimensional solution space, under the assumption of a pre-established Global Optima.

2.Accelerating PSO with Parallel Computations

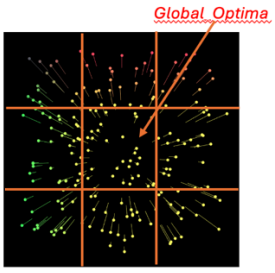


Figure1: Convergence of local optima for all swarm particles to a single global optima in a 2D-space. Picture Credit: PSO, Becky Johnson[Medium].

The process of computing the local_optima, encompassing both position and momentum for each particle within the PSO framework, represents a continuous nonlinear function. This function, illustrated in Figure 2, involves calculating the position of each particle relative to the current position (Pi(Xp,Yp)) in relation to the global optima (G(Xg,Yg)). Figure 1 showcases the iterative nature of this computation, illustrating how local_optimal is recalculated across iterations until the swarm converges on an assumed global solution. Given that this function is independently and iteratively computed for each particle, PSO stands to benefit significantly from the parallelization of computation. Moreover, this algorithm serves as an excellent candidate for analyzing the scaling behavior of parallel computation systems by varying parameters such as swarm size and distribution within the problem space. Such analyses provide valuable insights into the efficiency and effectiveness of parallel computing paradigms when applied to PSO algorithms.

3. Implementation of PSO using MPI

This project implements PSO using Message Passing Interface (MPI) for parallel computation to form asynchronous parallel particle swarm, solving for convergence problem.

For implementation we assume design parameters as described in section 3.2 and flow-chart delineating steps in figure3.

$$\vec{x}_i = G\left(\frac{\vec{p}_i + \vec{g}}{2}, ||\vec{p}_i - \vec{g}||\right) ,$$

Figure2: Bare-Bone PSO for particle position vector.

Implementing Particle Swarm Optimization (PSO) using Message Passing Interface (MPI) involves orchestrating a distributed computing environment where multiple processes collaborate to solve optimization problems in parallel. Initially, the MPI environment is initialized, delineating the master process responsible for coordination and the worker processes tasked with executing computations. Particle initialization follows, wherein positions and velocities are randomly assigned to particles across the processes. Subsequently, fitness evaluations are concurrently conducted within each process to determine the fitness value for each particle. Communication mechanisms are then employed to exchange information between processes, facilitating the identification of the global best solution. This necessitates synchronizing the best solutions found by each process and updating the global best accordingly. Following this, particle positions and velocities are iteratively updated based on PSO equations, with inter-process communication facilitating the exchange of information about the global best solution. Throughout the iterative process, termination criteria are evaluated to ascertain if further iterations are required.

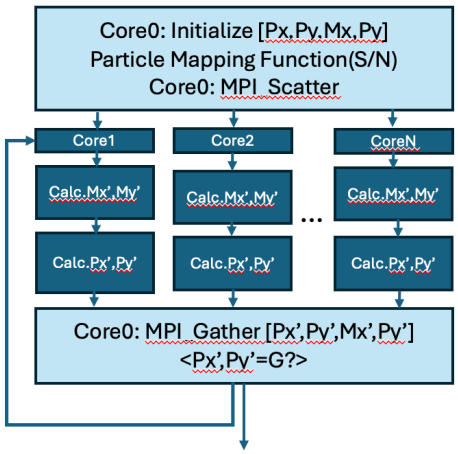


Figure3: Flow Chart for PSO using MPI Scatter/Gather

3.2 Assumptions for Algorithm Implementation

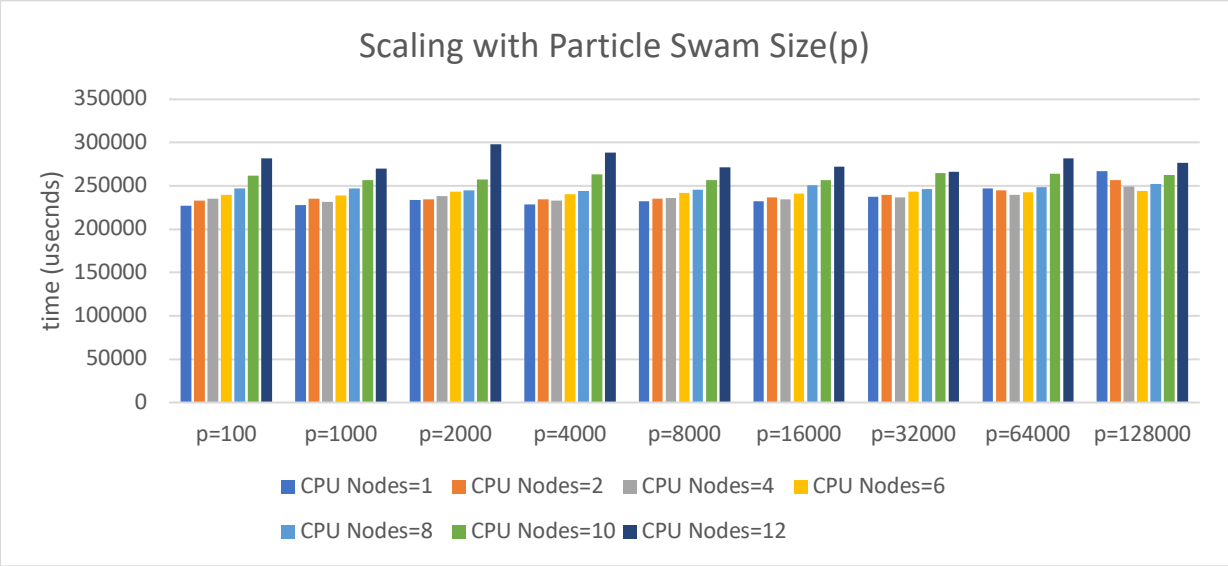
- Initialization:
  - Coordinates of particles are randomly chosen.
  - Global_optima is randomly located. (Xg,Yg).
  - Coordinates are within {0,2^32-1}, for integer calculations.
- Constant Acceleration:
  - All particles apply equal acceleration till G is reached. (s=ut+1/2.at^2)
- Iterations:
  - Maximum Iteration limit is kept to 10^6.
- Implemented in C/C++ semantics with cblas/gsl lib to accelerate math functions.

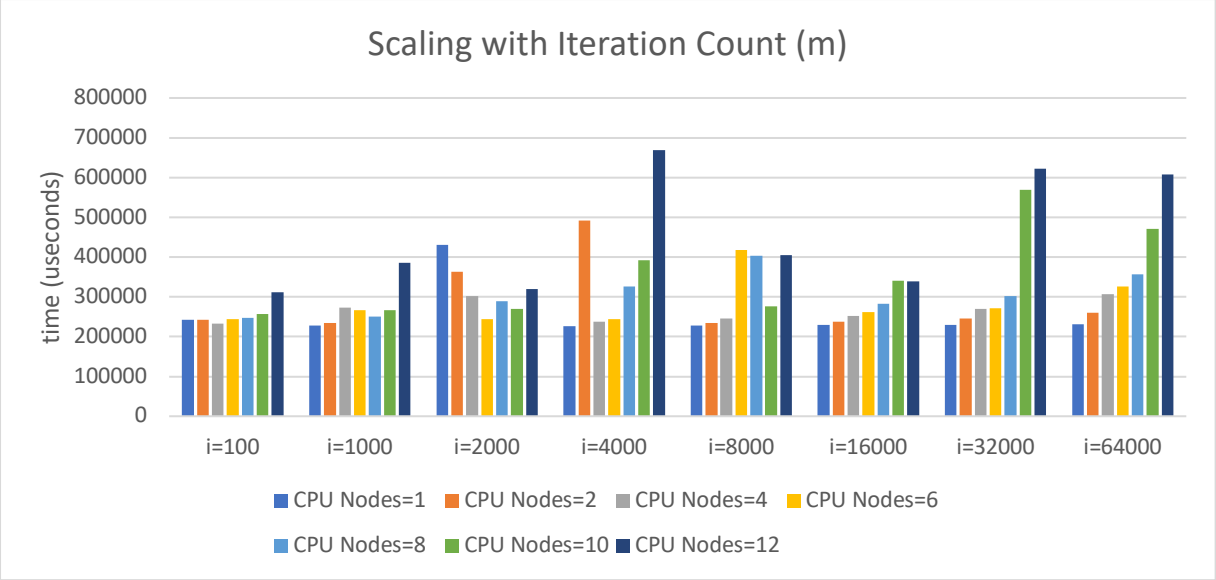
3.3 Snippet of PSO algorithm

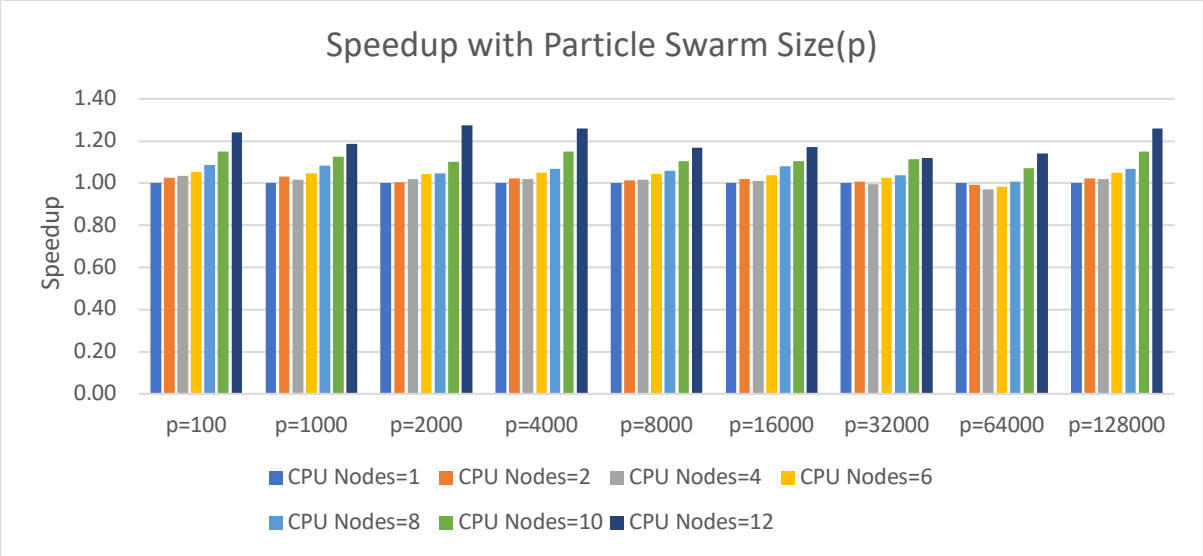
```
//------------------------
// PPSO calculation - local/global optima
//------------------------
for (step=0; step<nIterations; step++) {
    for (i=0; i<distributed_particles; i++) {
        for (j=0; j<nDimensions; j++) {
            // calculate stochastic coefficients
            rho1 = c1 * gsl_rng_uniform(r);
            rho2 = c2 * gsl_rng_uniform(r);
            // update velocity
            velocities[i][j] = w * velocities[i][j] + \
            rho1 * (pBestPositions[i][j] - positions[i][j]) +  \
            rho2 * (gBestPosition[j] - positions[i][j]);
            // update position
            positions[i][j] += velocities[i][j];

            if (positions[i][j] < x_min) {
                positions[i][j] = x_min;
                velocities[i][j] = 0;
            } else if (positions[i][j] > x_max) {
                positions[i][j] = x_max;
                velocities[i][j] = 0;
            }

        }

        // update particle optima
        fit = ackley(positions[i], nDimensions);
        // update personal best position?
        if (fit < pBestFitness[i]) {
            pBestFitness[i] = fit;
            // copy contents of positions[i] to pos_b[i]
            memmove((void *)&pBestPositions[i], (void *)&positions[i],
                sizeof(double) * nDimensions);
        }
        // update global optimia
        if (fit < gBestFitness) {
            // update best fitness
            gBestFitness = fit;
            // copy particle pos to gbest vector
            memmove((void *)gBestPosition, (void *)&positions[i],
                sizeof(double) * nDimensions);
        }
    }
//------------------------
    for(int k=0;k<(nDimensions);k++)
        sendingdata[k]=gBestPosition[k];

    sendingdata[(int)nDimensions]=gBestFitness;
    MPI_Gather(&sendingdata,nDimensions+1, MPI_INT,&recievingdata,nDimensions+1, MPI_INT, 0, MPI_COMM_WORLD);
```

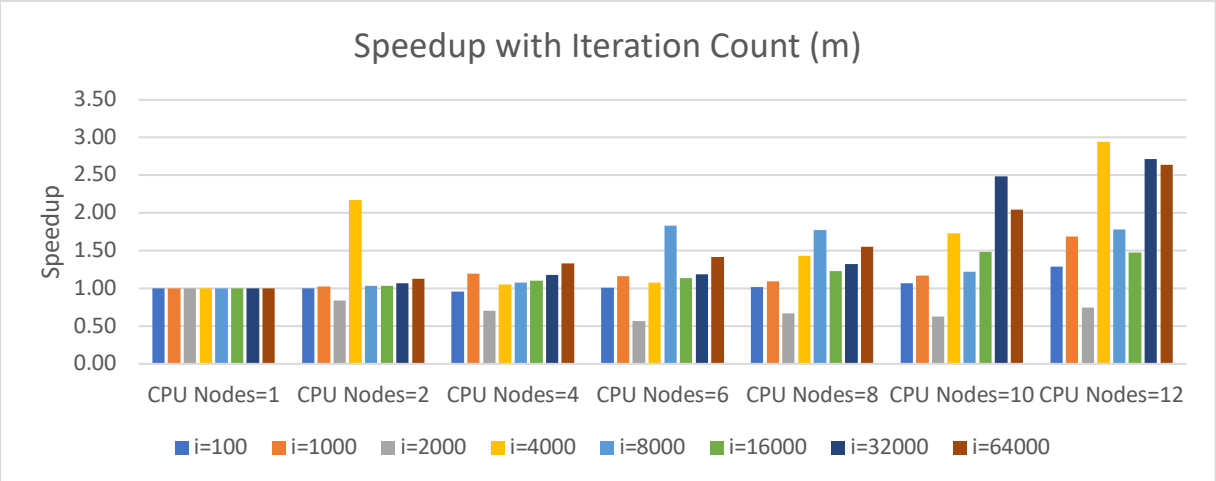
4. Evaluating PSO for Scaling and Speedup

For evaluation, this project scales parameters of PSO algorithm for swarm size and iteration count across number of CPU Node. In the figure below we show, scaling and Speedup of PSO across these parameters.




Scaling with Iteration Count (m)
time (useconds)
800000
700000
600000
500000
400000
300000
200000
100000
0
i=100
i=1000
i=2000
i=4000
i=8000
i=16000
i=32000
i=64000
CPU Nodes=1
CPU Nodes=2
CPU Nodes=4
CPU Nodes=6
CPU Nodes=8
CPU Nodes=10
CPU Nodes=12


Speedup with Particle Swarm Size(p)
Speedup
1.40
1.20
1.00
0.80
0.60
0.40
0.20
0.00
p=100
p=1000
p=2000
p=4000
p=8000
p=16000
p=32000
p=64000
p=128000
CPU Nodes=1
CPU Nodes=2
CPU Nodes=4
CPU Nodes=6
CPU Nodes=8
CPU Nodes=10
CPU Nodes=12


Speedup with Iteration Count (m)
Speedup
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
CPU Nodes=1
CPU Nodes=2
CPU Nodes=4
CPU Nodes=6
CPU Nodes=8
CPU Nodes=10
CPU Nodes=12
i=100
i=1000
i=2000
i=4000
i=8000
i=16000
i=32000
i=64000

5. Conclusion

Asynchronous Parallel PSO implemented using MPI in this project clearly highlights the scaling and speedup that can be achieved. For higher number of Nodes/threads, upto speed of 2.95x can be achieved for swarm of size of 1000. Similarly, we observed as the swarm size in increased from 100 to 128000, speedup of 30% is achieved. Similarly, for as the number of iteration increases from 100 to 64000 to converge global optima speedup of 195% is achieved with CPU count=12.

References

1. Kennedy, J.; Eberhart, R. (1995). "Particle Swarm Optimization". Proceedings of IEEE International Conference on Neural Networks. Vol. IV. pp. 1942–1948. doi:10.1109/ICNN.1995.488968.
2. A Gentle Introduction to Particle Swarm Optimization, by Adrian Tam on October 12, 2021. https://machinelearningmastery.com/
3. Luca Mussi, Fabio Daolio, and Stefano Cagnoni. 2011. Evaluation of parallel particle swarm optimization algorithms within the CUDATM architecture. Inf. Sci. 181, 20 (October 2011), 4642–4657. https://doi.org/10.1016/j.ins.2010.08.045
4. Chuan-Chi Wang, Chun-Yen Ho, Chia-Heng Tu, and Shih-Hao Hung. 2022. CuPSO: GPU parallelization for particle swarm optimization algorithms. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22). Association for Computing Machinery, New York, NY, USA, 1183–1189. https://doi.org/10.1145/3477314.3507142
5. https://en.wikipedia.org/wiki/Particle_swarm_optimization