OpenMP and MPI implementation of particle swarm optimization

Zhuowen Zhao
Chemical Engineering and Materials Science
Michigan State University
East Lansing, MI 48824, USA
zhaozhuo@msu.edu

***Abstract*—Particle swarm optimization (PSO) was implemented with Open Multi-Processing (OpenMP) and Message Passing Interface (MPI), respectively. The speedup performance of both parallel methods was evaluated by comparing it with that of a sequential implementation in both strong and weak scaling studies.**

***Index Terms*—PSO, parallel computing, OpenMP, MPI**

I. Introduction

Robust non-convex optimization algorithms are useful tools for computational modeling. Meaningful model parameters can be obtained by optimizing the parameters of interest based on experimental data through an inverse approach [1, 2]—iteratively updating parameters until a "minimum" deviation between simulated and experimental data is achieved. The optimization process can be computationally costly, depending on the complexity of the problems.

Particle swarm optimization (PSO) is a non-derivative optimization algorithm that was introduced about 30 years ago [3]. An advantage of PSO over other optimization algorithms is its parallelizable "nature". The exploration efficiency of the parameter space is directly related to the number of particles (N_{particle}), and the ideal speedup is N_{particle} if every particle is updated simultaneously in each iteration. Therefore, a parallel implementation may improve the speed of PSO.

This report aims to investigate the speedup performance of PSO with Open Multi-Processing (OpenMP) and Message Passing Interface (MPI) implementations by comparing with that of the sequential implementation.

II. Methods

A. Particle swarm optimization algorithm (PSO)

PSO was firstly introduced by [3]. It mimics the idea of unpredictable choreography of a bird flock and is further developed for multi-dimensional search with acceleration by distance [4]. Pre-assigned number of particles N_{particle} (potential solutions) are used to explore the parameter space (normalized $N_{\text{parameter}}$ space). Particles are initialized at random locations and velocities. The trajectory of an individual particle is only affected by the two locations (called "tractors") that give the best history fitness for itself (pbest) and among all particles (gbest); see Eq. (1). The maximum magnitude of velocity for each particle is limited to a cut-off value, v_{max}. Each particle also has a random inertial factor (w) that carries the momentum from the previous motion. During the iterations, the tractors will be updated to new locations (respectively) if better fitnesses values are found.

$$v_{\text{i}} = w_{\text{i}} * v_{\text{i}} + c_1 * \text{ rand}() * (\text{pbest}_{\text{i}} - x_{\text{i}}) + c_2 * \text{rand}() * (\text{gbest } - x_{\text{i}}) \quad (1)$$

$$x_{\text{i}} = x_{\text{i}} + v_{\text{i}} \quad (2)$$

The parameters for the PSO used are $c_1 = c_2 = 1.494$, inertia factor $w_{\text{i}} = 0.5 + 0.5 * rand()$, $v_{\text{max}} = 1$, respectively, which are suggested by [4]. The tractors pbest_i and gbest are initialized with infinity.

B. Parallel implementation

PSO is similar to Nbody problem in terms of general implementation. However, the complexity of one iteration (update) is $O(n)$ instead of $O(n^2)$ because each particle update (location and velocity) is not dependent on the whole population but only based on the global best particle (smallest historical fitness in the group). One can understand PSO as a simpler version of Nbody problem.

OpenMP threading Particles (location and velocity) are stored in arrays ($N_{\text{dimension}}$x N_{particle}). The implementation updates each particle in *OpenMP for* loops. OpenMP can "spawn" multiple threads to accelerate these loop with `#pragma omp parallel for`. Because the computation load is more or less the same for calculating each particle, `schedule(static)` is used for managing load balance among different threads [5, 6].

OpenMP threading should be avoided when updating the global best particle because shared memory access could cause a racing condition that leads to false sharing of the global best particle. There might be some ways to take care of this situation using OpenMP, but they are not considered in the present work.

"Domain decomposition" with MPI The concept analogous to domain decompositions in molecular dynamics and Nbody problem [5, 7] is used in PSO MPI implementation. The idea is to let each rank (processor) only work on a subgroup (equal amount if possible for load balancing among processors) of the total particle population within each iteration. After all

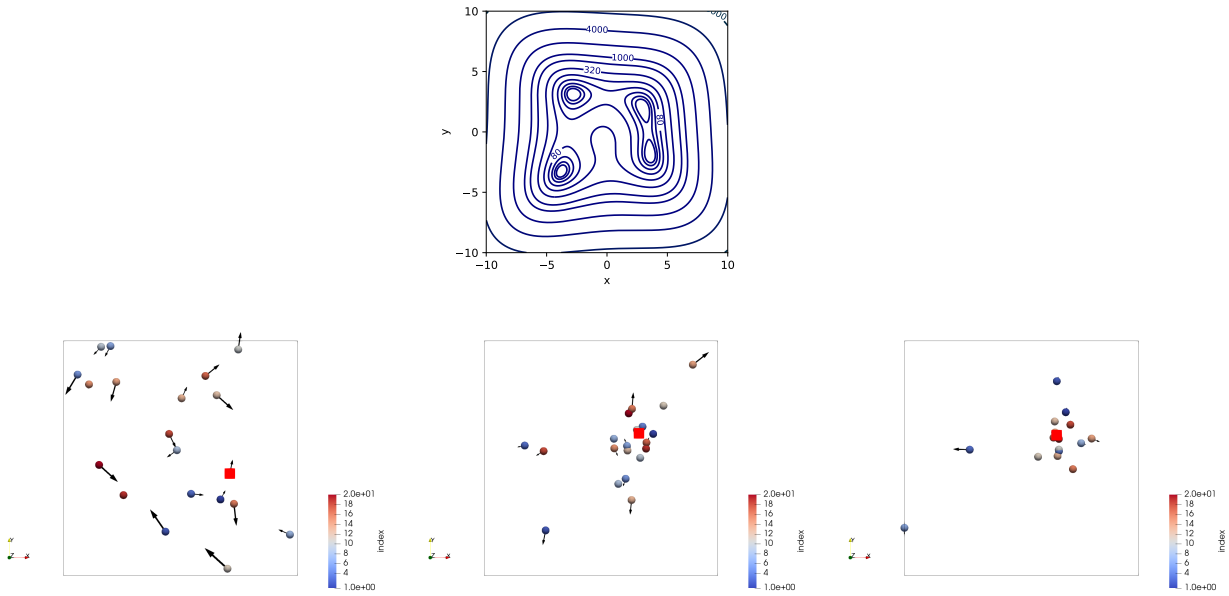

Fig. 1: Top row: Himmelblau's function has 4 identical local minima shown in contour plots. Bottom row: schematics of finding one of the local minima of Himmelblau's function (top right one in contour plots, see red square in bottom figures which is the global best particle/solution) using Particle Swarm Optimization in 25 iterations (N_{particle} = 20, $N_{\text{iteration}}$ = 0, 10, 25 from left to right, bounds are [-10,10] for both dimensions).

processors finish their work, the newly updated information is collected and broadcast to all processors before the next iteration with `MPI_Allgather` command. To ensure synchronization, the update of global best location is done by the master rank and is broadcast to all the other ranks before the next iteration.

C. HDF5 I/O

Due to the high volume of spatial data, HDF5 is the preferred output method to keep track of particle locations and velocities for each iteration. The fitness for each iteration and average time cost (total) are printed in the terminal.

III. Verification with Himmelblau's Function

Himmelblau's function is a simple two-dimensional function often used to test mathematical optimization algorithms. Himmelblau's function [8] is defined by Eq. (3). It has four identical local minima ($f(\cdot) = 0$) or "solutions". They are listed in Table I and can be visualized as four concentric circles in the contour plots in Fig. 1.

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2 \tag{3}$$

Subfigures in the bottom row in Fig. 1 demonstrate how 20 PSO particles find one of the correct solutions (red square, the global best particle) in 25 iterations. With Himmelblau's function, PSO implementation with either OpenMP or MPI is verified by checking the optimized results with the correct solutions. Table I also shows the optimized results by PSO with OpenMP. It usually takes a few hundred iterations to find solutions whose fitness < 0.05 (less than a $5.000\,\%$ deviation compared to the correct minima).

TABLE I: Four local minima of Himmelblau's function and the optimized results by PSO with OpenMP (fitness < 0.05, bounds are [-20, 20] for both dimensions).

minima (x,y)	omp optimized (x,y)	iterations	fitness
3.000, 2.000	2.966, 2.039	301	0.041
−2.805, 3.131	−2.836, 3.124	140	0.033
−3.779, −3.283	−3.762, −3.257	307	0.035
3.584, −1.848	3.574, −1.798	131	0.038

IV. Results and Discussions

In order to compare the performance of different parallel strategies, a fixed iteration number ($N_{\text{iteration}}$=100) is used for all cases. To minimize fluctuations, the averaged time cost of 10 repetitive runs is used as the metric.

Strong scaling and weak scaling studies are done for both parallel methods. Strong scaling studies the speedup of a fixed-size problem with varying processor numbers, which is governed by Amdahl's law. In comparison, weak scaling investigates a scaled size problem with a fixed workload per processor and obeying Gustafson's law [5, 6]. The efficiency of strong scaling E_{strong} and weak scaling E_{weak} are calculated in Eq. (4) and Eq. (5). Both E_{strong} and E_{weak} are expected to follow an asymptotic decay due to increasing overhead cost with more processor/threads used.

$$E_{\text{strong}} = \frac{T_1}{N_{\text{processor}} \times T_n} \tag{4}$$

$$E_{\text{weak}} = \frac{T_1}{T_n} \tag{5}$$

All jobs are submitted on Intel18-dev node on ICER High Performance Computing clusters at Michigan State University.

A. Strong scaling of OpenMP implementation (thread-to-thread speedup)

N_{particle}=100, 8000, and 16000 are fixed for `OMP_NUM_THREADS`=1, 2, 4, 8, 16, 32, and 40 respectively. The performance are plotted in Fig. 2. For N_{particle}=100 and 8000, the time cost slightly increases with more threads spawned. This is because the fitness calculations (Himmelblau's function) are small, so the overhead cost from spawning more threads time outweighs the cost saved by parallel executions. When N_{particle} is increased to 16000, the time cost starts to slightly decrease with more threads. Unfortunately, N_{particle}=16000 is the max particle number the default memory could handle, so it could not be tested further. It is expected to see time cost would reduce more significantly with increasing thread numbers for $N_{\text{particle}}>$ 16000.

The efficiency of every N_{particle} follows an asymptotic decay, which is expected.

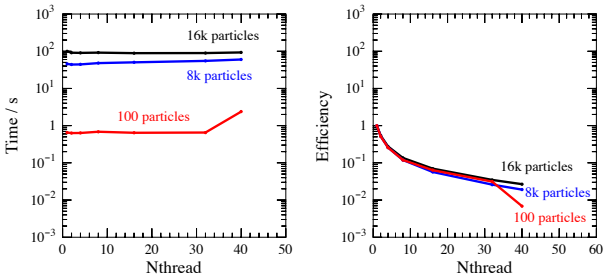

Fig. 2: Strong scaling study of OpenMP PSO for N_{particle}=100, 8000, and 16000 (max particles default memory can handle).

B. Weak scaling of OpenMP implementation

Since PSO has $O(n)$ complexity, a linear scale-up of N_{particle} with N_{thread} is used to maintain the local workload at 400 and 1000 particles/thread, respectively. In an ideal scenario (no extra cost associated with spawning threads), if the problem is perfectly parallelizable, a flat line (time cost) is expected to see. However, Fig. 3 shows a significant overhead cost associated with spawning more threads compared to the actual calculation cost such that the time cost increases with more threads (efficiency decreases) for this problem. On the other hand, efficiency also drops asymptotically with increasing thread number.

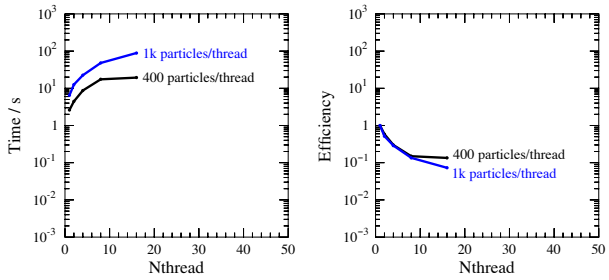

Fig. 3: Weak scaling study of OpenMP PSO for work load of 400 and 1000 particles per thread.

C. Strong and weak scaling of MPI implementation

Despite the MPI code can get the correct solutions for $N_{\text{processor}}$=2, it cannot run when $N_{\text{processor}} > 2$ for some reasons (return segmentation 11 error). This may due to the way memory is allocated in the present implementation. Therefore, I can only show results for $N_{\text{processor}}$=1, 2 and up to N_{particle}=1000 (max particles for MPI implementation). Due to the limited data points, not much generalization can be made about the performance except two take-aways: (a) time cost increases a lot as $N_{\text{processor}}$ increases to 2; (b) MPI implementation has arguably more time cost compared to OpenMP for both strong and weak scaling studies. This might be MPI implementation has a lot more cost associated with communication (`MPI_Bcast` and `MPI_Allgathter` among all processors) compared to OpenMp thread spawning.

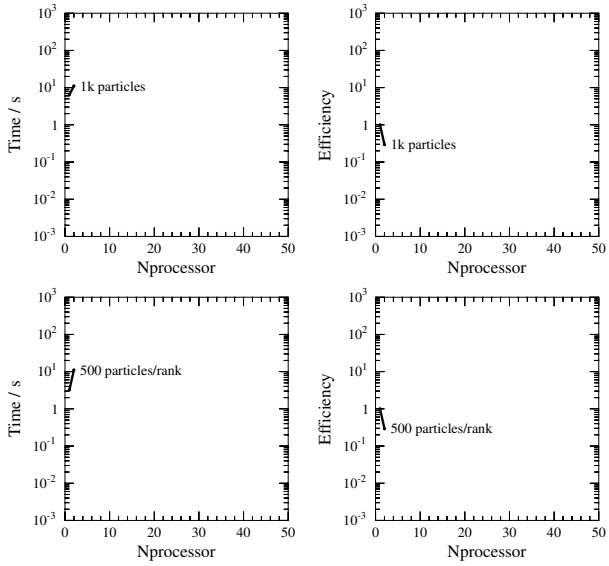

Fig. 4: Strong (top row) and weak (bottom) scaling study with MPI.

D. Load balancing

As is discussed in Section II-B, `#pragma parallel omp for schedule(static)` is used to ensure every thread has more or less the same workload during OpenMP

threading. Since the cost to calculate the fitness value for each particle update is almost the same, an equal number of particles are distributed to each processor in MPI parallelization to balance the workload among processors.

E. Memory usage

In the sequential implementation, all variables are defined as global variables to ensure it does not need to duplicate data when updating particles. However, the max particle number it could handle (with default memory) is N_{particle}=16000. More memory could be requested through Slurm to allow more particle numbers.

For OpenMP implementation, memory is not a big issue as threads use shared memory. MPI implementation would have more memory limitation because each processor needs to duplicate the entire dataset, and the memory burden scales up with $N_{\text{processor}}$ (max N_{particle}=1000 for MPI implementation). A future improvement to address this issue is to pass data only relating to the particle subgroup to each processor.

V. Summary

Parallel computing implementation of particle swarm optimization (PSO) was developed with OpenMP and MPI, respectively. The speedup performance of both parallel methods was evaluated by comparing it with that of a sequential implementation in both strong and weak scaling studies. The performance did not show a significant speedup for both parallel methods. This indicates PSO is not complicated enough (when fitness calculation is also cheap) to use either parallel scheme (OpenMP or MPI). A hybrid parallel paradigm (OpenMP + MPI) needs further investigation. However, when the fitness calculation requires enormous efforts (such as finite element modeling), parallel computing is expected to have considerable speedup. It would also be interesting to test this in future practice.

Acknowledgement

This is a term report for the parallel computing course offered by the Department of Computational Mathematics, Science, and Engineering, at Michigan State University. The author greatly appreciates the inspiration and help from professor Sean M. Couch. The computational calculation of this work was supported by ICER High Performance Computing Clusters (HPCC) at Michigan State University.

Code availability

The source codes can be found at https://github.com/zhuowenzhao/particle_swarm_parallel_implementation.

References

[1] Z. Zhao, "The influence of anisotropic slip and shear transformation on heterogeneous deformation based upon nanoindentation, crystal plasticity modeling, and artificial neural networks," Ph.D. dissertation, Michigan State University, 2021.

[2] Z. Zhao, M. R. Ruiz, J. Lu, M. A. Monclus, J. M. Molina-Aldareguia, T. R. Bieler, and P. Eisenlohr, "Quantifying the uncertainty of critical resolved shear stress values derived from nano-indentation in hexagonal ti alloys," *Experimental Mechanics*, vol. 62, pp. 731–743, 2022.

[3] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4, 1995, pp. 1942–1948 vol.4.

[4] E. Ozcan and C. K. Mohan, "Particle swarm optimization: surfing the waves," in *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, vol. 3, 1999, pp. 1939–1944 Vol. 3.

[5] V. Eijkhout, R. van de Geijn, and E. Chow, *Introduction to High Performance Scientific Computing*. Zenodo, apr 2016, Book source and printed copies are available: http ://pages.tacc.utexas.edu/ eijkhout/istc/istc.html. [Online]. Available: https://doi.org/10.5281/zenodo.49897

[6] V. Eijkhout, *Parallel Programming for Science Engineering*. Zenodo, 2016. [Online]. Available: https://pages.tacc.utexas.edu/~eijkhout/pcse/html/index.html

[7] D. Bindel, "N-body assignment," Feb 2010. [Online]. Available: https://www.cs.cornell.edu/~bindel/class/cs5220-s10/hw2code.pdf

[8] H. D. Mautner, *Applied Nonlinear Programming [Texte Imprime]*. McGraw-Hill, 1972. [Online]. Available: https://ulysse.univ-lorraine.fr/discovery/fulldisplay/alma991002153909705596/33UDL_INST:UDL