Noah Robinson
Jack Pinkstone
Professor Kandasamy
ECEC 413
24 May 2023

Particle Swarm Optimization OMP Report

This assignment required the students to use the OMP library to write multi-threaded code to parallelize the particle swarm process. We parallelized the solve gold function, the best fitness function behavior, and the pso_init function.

Design

PSO Solve OMP

```
int solve_omp(char *function, swarm_t *swarm, float xmax, float xmin, int
max_iter, int num_threads)
{
    int i, j, iter, g;
    float w, c1, c2;
    float r1, r2;
    float curr_fitness;
    particle_t *particle, *gbest;

    w = 0.79;
    c1 = 1.49;
    c2 = 1.49;
    iter = 0;
    g = -1;

    int g_local;
    float best_fitness = INFINITY;
    float local_fitness;

    unsigned int seed = time(NULL); // Seed the random number generator
    while (iter < max_iter) {
#pragma omp parallel num_threads(num_threads) private(i, j, r1, r2, particle,
gbest, curr_fitness, local_fitness, g_local) shared(best_fitness, g)
{
#pragma omp for
            for (i = 0; i < swarm->num_particles; i++) {
                particle = &swarm->particle[i];
                gbest = &swarm->particle[particle->g];  /* Best performing
particle from last iteration */

                // #pragma omp parallel for private(j, r2, r1) shared(particle,
gbest, xmax, xmin) schedule(static) num_threads(num_threads)
                for (j = 0; j < particle->dim; j++) {   /* Update this particle's
state */
                    r1 = (float)rand_r(&seed)/(float)RAND_MAX;
                    r2 = (float)rand_r(&seed)/(float)RAND_MAX;
                    /* Update particle velocity */
                    particle->v[j] = w * particle->v[j] + c1 * r1 *
(particle->pbest[j] - particle->x[j]) + c2 * r2 * (gbest->x[j] - particle->x[j]);
                    /* Clamp velocity */
                    if ((particle->v[j] < -fabsf(xmax - xmin)) || (particle->v[j]
> fabsf(xmax - xmin)))
                        particle->v[j] = uniform(-fabsf(xmax - xmin), fabsf(xmax
- xmin));

                    /* Update particle position */
                    particle->x[j] = particle->x[j] + particle->v[j];
                    if (particle->x[j] > xmax)
                        particle->x[j] = xmax;
                    if (particle->x[j] < xmin)
                        particle->x[j] = xmin;
                } /* State update */

                /* Evaluate current fitness */
                pso_eval_fitness(function, particle, &curr_fitness);

                /* Update pbest */
                if (curr_fitness < particle->fitness) {
                    particle->fitness = curr_fitness;
                    for (j = 0; j < particle->dim; j++)
                        particle->pbest[j] = particle->x[j];
                }
            } /* Particle loop */

                /* Identify best performing particle */
                //g = pso_get_best_fitness_omp(swarm, num_threads);
            local_fitness = INFINITY;
            g_local = -1;
#pragma omp for
            for (i = 0; i < swarm->num_particles; i++) {
                particle = &swarm->particle[i];
                if (particle->fitness < local_fitness) {
                    local_fitness = particle->fitness;
                    g_local = i;
                }
            }
#pragma omp critical
{
            if(local_fitness < best_fitness)
            {
                g = g_local;
                best_fitness = local_fitness;
            }
}

#pragma omp barrier
            // #pragma omp parallel for private(i) shared(swarm, g)
num_threads(num_threads)
#pragma omp for
            for (i = 0; i < swarm->num_particles; i++) {
                particle = &swarm->particle[i];
                particle->g = g;
            }
}
            iter++;
        } /* End of iteration */

    return g;
}
```

The code above is an OpenMP function that runs in each thread for the omp solve function. During each PSO iteration, the for loop within the pso solve gold() function was parallelized using OpenMP to increase the particle swarm's overall performance. This was done in order to speed up the execution of the code by allowing multiple threads to run simultaneously.

Lastly, we initialized the code for the get best fitness function so that it does not create a chokepoint. For this we find local (aka partial) best fitnesses and see if they are better than the global best fitness at the end in a critical section.

Parallel PSO Init

```c
swarm_t *pso_init_omp(char *function, int dim, int swarm_size,
                  float xmin, float xmax, int num_threads)
{
    int i, j, g;
    int status;
    float fitness;
    swarm_t *swarm;
    particle_t *particle;

    swarm = (swarm_t *)malloc(sizeof(swarm_t));
    swarm->num_particles = swarm_size;
    swarm->particle = (particle_t *)malloc(swarm_size * sizeof(particle_t));
    if (swarm->particle == NULL)
        return NULL;

#pragma omp parallel num_threads(num_threads) private(particle, status, fitness)
{
    #pragma omp for
    for (i = 0; i < swarm->num_particles; i++) {
        particle = &swarm->particle[i];
        particle->dim = dim;
        /* Generate random particle position */
        particle->x = (float *)malloc(dim * sizeof(float));
        for (j = 0; j < dim; j++)
           particle->x[j] = uniform(xmin, xmax);

       /* Generate random particle velocity */
        particle->v = (float *)malloc(dim * sizeof(float));
        for (j = 0; j < dim; j++)
            particle->v[j] = uniform(-fabsf(xmax - xmin), fabsf(xmax - xmin));
        /* Initialize best position for particle */
        particle->pbest = (float *)malloc(dim * sizeof(float));
        for (j = 0; j < dim; j++)
            particle->pbest[j] = particle->x[j];

        /* Initialize particle fitness */
        status = pso_eval_fitness(function, particle, &fitness);
        if (status < 0) {
            fprintf(stderr, "Could not evaluate fitness. Unknown function 
provided.\n");
            exit(0);
        }
        particle->fitness = fitness;

        /* Initialize index of best performing particle */
        particle->g = -1;
    }
#pragma omp single
    /* Get index of particle with best fitness */
    g = pso_get_best_fitness_omp(swarm, num_threads);
#pragma omp for
    for (i = 0; i < swarm->num_particles; i++) {
        particle = &swarm->particle[i];
        particle->g = g;
    }
}
    return swarm;
}
```

The code above is an OpenMP function that runs in each thread for the PSO INIT function. The old init function used to initialize the swarm in single-threaded fashion, but with OpenMP Parallelization, we can process the function faster. This is because OpenMP allows multiple threads to run simultaneously, which can speed up the execution of the code. In this case, each thread will initialize a portion of the swarm, which will reduce the overall time it takes to initialize the entire swarm.

Performance Comparison

Rastrigin function:

```
./pso rastrigin 10 3000 -5.12 5.12 950 #
```

4 Threads
Gold Time: 2.922961
OMP Time: 2.066142

8 Threads
Gold Time: 2.522413
OMP Time: 2.259342

16 Threads
Gold Time: 2.834921
OMP Time: 2.424863

```
./pso rastrigin 10 10000 -5.12 5.12 10000 #
```

4 Threads
Gold Time: 63.798714
OMP Time: 71.268349

8 Threads
Gold Time: 59.189140
OMP Time: 80.385231

16 Threads
Gold Time: 67.040649
OMP Time: 87.996361

Schwefel function:

```
./pso schwefel 20 1000 -500 500 10000 #
```

4 Threads
Gold Time: 59.796974
OMP Time: 24.994535

8 Threads
Gold Time: 486.032074

OMP Time: 175.257431

16 Threads
Gold Time: 549.691956
OMP Time: 227.627823

The OpenMP version of the code did not have much of a difference in execution time for the easier functions. However, for the more challenging functions with significantly more computation, the OpenMP code significantly outperformed the sequential code for the schwefel function but seemed to perform worse than the rastrigin function when using 10,000 and 10,000. This is because the parallel version can distribute the work across multiple threads, which can significantly reduce the execution time.

In other words, the OpenMP version of the code can break down the work into smaller tasks that can be executed simultaneously by multiple threads. This allows the code to finish executing much faster than if it were to be executed sequentially by a single thread.