Ehi Simon - 14352307
Alisha Augustin - 14389451

ECEC 413-001
05/21/2023

Open MP: Particle Swarm Optimization

Particle swarm optimization optimizes the code by initializing a population of random solutions called particles over the desired search space; and then by moving these particles around the search space in iterative fashion until the best solution is found. Our completed code is a parallel implementation of the PSO algorithm using OpenMP directives.

The first function we included was *uniform_omp*, which differs from the original *uniform* function because we used rand_r with a unique seed. The *uniform_omp* function ensures that each thread has its own independent random number generator, making it safe for parallel execution. This seed is generated by *time(NULL)* and is used to calculate the normalized variable.

```
float uniform(float min, float max)
{
    float normalized;
    normalized = (float)rand()/(float)RAND_MAX;
    return (min + normalized * (max - min));
}

float uniform_omp(float min, float max)
{
    float normalized;
    unsigned int seed = time(NULL);
    normalized = (float)rand_r(&seed)/(float)RAND_MAX;
    return (min + normalized * (max - min));
}
```

Figure 1: Parallelization of uniform function

The function *pso_get_best_fitness* has a parallelized version called *pso_get_best_fitness_omp*. This function determines the index of the particle with the best fitness among all particles in the swarm. The parallel version divides the work among multiple threads, each responsible for evaluating a subset of particles, and then finds the best fitness in a parallel reduction fashion. The code for both functions can be found in the figure below.

```c
/* Return index of best performing particle */
int pso_get_best_fitness(swarm_t *swarm)
{
    int i, g;
    float best_fitness = INFINITY;
    particle_t *particle;

    g = -1;
    for (i = 0; i < swarm->num_particles; i++) {
        particle = &swarm->particle[i];
        if (particle->fitness < best_fitness) {
            best_fitness = particle->fitness;
            g = i;
        }
    }
    return g;
}

int pso_get_best_fitness_omp(swarm_t *swarm)
{
    int g = -1;
    float best_fitness = INFINITY;

    #pragma omp parallel shared(g, best_fitness)
    {
        particle_t *particle;

        #pragma omp for
        for (int i = 0; i < swarm->num_particles; i++) {
            particle = &swarm->particle[i];
            if (particle->fitness < best_fitness) {
                #pragma omp critical
                {
                    best_fitness = particle->fitness;
                    g = i;
                }
            }
        }
    }

    return g;
}
```

Figure 2: Parallelization of pso_get_best_fitness function

Lastly, our *pso_init* function is responsible for initializing the swarm. It takes parameters such as the optimization function to be used, the dimensionality of the search space, the swarm size, and the range of values for particle positions.

The function allocates memory for the swarm structure and the particles within it. It then initializes each particle by generating random positions and velocities within the specified range. The fitness of each particle is evaluated using the provided optimization function, and the best performing particle index (g) is determined.

The *pso_init_omp* function is the parallelized version of *pso_init* using OpenMP. It follows a similar initialization process but distributes the work of initializing particles among multiple

threads using the #pragma omp parallel for directive to parallelize the for loop where we create space in memory to generate random particle position, random velocity, initialize best position for particle and initialize the best fitness.

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

        #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(-1);
                //return NULL;
            }
            particle->fitness = fitness;

            /* Initialize index of best performing particle */
            particle->g = -1;
        }
```

Figure 3: Parallelization of pso_init function

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

    return swarm;
}
```

Figure 4: #pragma omp master directive in pso_init_omp function

Performance (Execution Time) Comparison

	Serial				Parallel			
	4 Threads	**8 Threads**	**16 Threads**	**32 Threads**	**4 Threads**	**8 Threads**	**16 Threads**	**32 Threads**
Rastrigin	8.085730 s	8.368644s	8.045931s	7.751763s	2.145487s	1.284453s	1.026542s	1.158271s
Schwefel	14.331111s	13.790656 s	14.443622s	14.441278 s	3.452430s	1.766760s	1.299258s	1.428037s

Parameters used:

Rastrigin: Dimension of 25, 1000 particles, minimum of -5.12, maximum of 5.12, 5000 iterations

Schwefel: Dimension of 25, 1000 particles, minimum of -500, maximum of 500, 5000 iterations

The table above is comparing the execution times between the serial and parallel implementation. Based on the results, the parallel implementation has a faster execution time as the number of threads increase.