**Antenna Arrays**

Antennas with a given radiation pattern may be arranged in a pattern (line, circle, plane, etc.) to yield a different radiation pattern.

*Antenna array* - a configuration of multiple antennas (elements) arranged to achieve a given radiation pattern.

*Linear array* - antenna elements arranged along a straight line.

*Circular array* - antenna elements arranged around a circular ring.

*Planar array* - antenna elements arranged over some planar surface (example - rectangular array).

*Conformal array* - antenna elements arranged to conform to some non-planar surface (such as an aircraft skin).

There are several array design variables which can be changed to achieve the overall array pattern design.



Array Design Variables

1. General array shape (linear, circular, planar, etc.).

2. Element spacing.

3. Element excitation amplitude.

4. Element excitation phase.

5. Patterns of array elements.

*Phased array* - an array of identical elements which achieves a given pattern through the control of the element excitation phasing.

Phased arrays can be used to steer the main beam of the antenna without physically moving the antenna.

Given an antenna array of identical elements, the radiation pattern of the antenna array may be found according to the *pattern multiplication* *theorem*.

Pattern multiplication theorem: vvv

***Array Pattern = Array Element Pattern x Array Factor (AF)***

*Array element pattern* - the pattern of the individual array element.

*Array factor* - a function dependent only on the geometry of the array and the excitation (amplitude, phase) of the elements.

**N-Element Linear Array**

The array factor *AF* is independent of the antenna type assuming all of the elements are identical. Thus, isotropic radiators may be utilized in the derivation of the array factor to simplify the algebra. The field of an isotropic radiator located at the origin may be written as (assuming θ-polarization)

We assume that the elements of the array are uniformly-spaced with a separation distance *d*.

In the far field of the array

.

.

.

The current magnitudes the array elements are assumed to be equal and the current on the array element located at the origin is used as the phase reference (zero phase).

The far fields of the individual array elements are

**.**

**.**

**.**

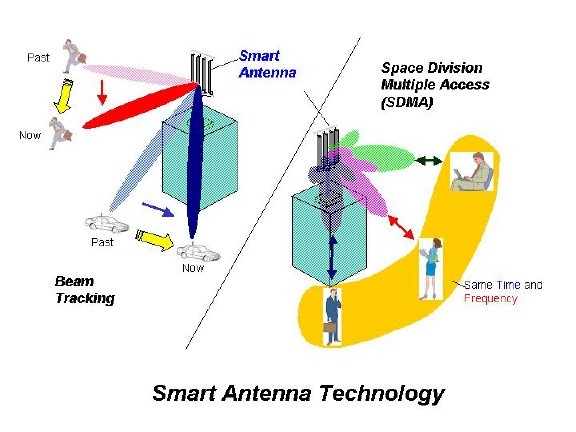
The overall array far field is found using superposition

**(Array factor for a uniformly-spaced *N*-element linear array)**

**Smart antennas**

(Also known as adaptive array antennas, multiple antennas and recently MIMO) are antenna arrays with smart signal processing algorithms used to identify spatial signal signature such as the direction of arrival (DOA) of the signal, and use it to calculate beam forming vectors, to track and locate the antenna beam on the mobile/target. The antenna could optionally be any sensor [5].

In many applications it is necessary to design antennas with very directive characteristics to meet the demands of long distance communication [1].

****

4.2 PSO Algorithm

http://www.swarmintelligence.org/

Particle swarm optimization (PSO) is a population based stochastic optimization technique developed by Dr. Eberhart and Dr. Kennedy in 1995, inspired by social behavior of bird flocking or fish schooling.

4\_108-7.pdf

Recently, the PSO technique has been successfully applied to the design of antennas and microwave components [6-9]. PSO is similar in some ways to Genetic Algorithms (GA) and otherevolutionary algorithms, but requires less computational bookkeeping in additions that the basic algorithm is easy to understand and implement.

****

**Figure 2.** Particle Swarm Optimization as modelled by a swarm of bees searching for flowers [7].

**PSO Basic Algorithm:**

Let’s assume an optimization problem that requires the optimization of a specific fitness function which depends on *M* variables. A collection or swarm of particles is distributed in the problem space, where each particle is assigned a random position in the *M*-dimensional problem space so that each particle position is a candidate solution to the optimization problem. Each of these particle positions is scored with a scalar cost to reflect how well it solves the problem. Then these particles fly to new positions through the *M*-dimensional problem space using both deterministic and stochastic update rules.

As the particles pass through the problem hyperspace, each particle remembers its own personal best position that it has ever found, called its local best or personal best *p*best and each particle also knows the best position found by any particle in the swarm, called the global best *g*best. On successive iterations, particles are pushed toward these prior best solutions.

This process can be visualized as a swarm of bees in a field [7]. Their goal is to find the location with the highest density of flowers. Without any knowledge of the field, the bees begin in random locations with random velocities looking for flowers. Each bee can remember the locations where it found the most flowers, and somehow knows the locations where the other bees found a plenty of flowers. Along the way, a bee might find a place with a higher concentration of flowers than it had found previously. It would then be drawn to this new location as well as the location of the most flowers found by the whole swarm. Occasionally, one bee may fly over a place with more flowers than had been encountered by any bee in the swarm. The whole swarm would then be drawn toward that location in additional to their own personal discovery. In this way, the bees explore the field.

Soon, all the bees swarm around this point.

The following steps are accomplished on each particle individually (see Fig. 3):

1- Initialize particles with random positions and velocities in *M* dimensions in the problem space.

2- For each particle, evaluate the desired optimization (fitness) function in *M* variables.

3- Update the particle velocity. The velocity of the particle is changed according to the relative locations of *p*best and *g*best . It is accelerated in the directions of these locations of greatest fitness according to the following equation [6]:

(6)

where is the velocity of the particle in the *n*-th dimension and iteration t and is the particle coordinate in the *n*-th dimension and iteration t, *c*1 and *c*2 are scaling factors that determine the relative pull of *p*best and *g*best (previous work has shown that a value of 2.0 is a good choice for both parameters [7]), and rand() is a random number uniformly distributed in interval (0,1). The parameter *w* is a number, called the inertial weight, in the range [0,1], which specifies the weight by which the particle current velocity depends on its previous velocity and how far the particle is from its personal best and global best positions.

4- Move the particle. Once the velocity has been determined, it is simple to move the particle to its next location. The new coordinate is computed for each of the dimensions according the following equation

*.*

5- Loop to step (2) until a criterion is met, usually a sufficiently good fitness or a maximum number of iterations.

**Our Proposed Modification:**

Here is an explanation of our proposed modification in PSO algorithm which are

1. **Use iteration best instead of personal best:** As the particles pass through the problem hyperspace, each particle knows the best position found by any particle in the swarm in this iteration, called the iteration best *iter*best instead of remembering its personal best. Also each particle knows the best position found by any particle in the swarm, called the global best *g*best. On successive iterations, particles are pushed toward these prior best solutions. And this modification shows better results than using of personal best.
2. **Modify in parameter’s values of c1, c2 and w:** This modification make c1, c2 and w change linearly with iterations according to the following equations:

w=wmax-((wmax-wmin)/itermax)\*ii,

c1=c1max-((c1max-c1min)/itermax)\*ii,

c2=c2min+((c2max-c2min)/itermax)\*ii,

where wmax=0.9, wmin=0.4, itermax=0.75 \* total no of iterations, ii is the current iteration number, c1max=2.5, c1min=0.5, c2max=2.5, c2min=0.5. All these parameter’s values are empirical values that we try and found these values was get the best results.

**Our proposed Sequential PSO:**

Here we will present the steps of our proposed PSO algorithm after the modifications done.

The following steps are accomplished on each particle individually (see Fig. 3):

1- Initialize particles with random positions and velocities in *M* dimensions in the problem space.

2- For each particle, evaluate the desired optimization (fitness) function in *M* variables.

3- Update the particle velocity. The velocity of the particle is changed according to the relative locations of *iter*best and *g*best . It is accelerated in the directions of these locations of greatest fitness according to the following equation [3]:

(4)

where is the velocity of the particle in the *n*-th dimension and iteration t and is the particle coordinate in the *n*-th dimension and iteration t, *c*1 and *c*2 are scaling factors that determine the relative pull of *iter*best and *g*best (our work has shown that a changing their value from 2.5 to 0.5 is a good choice for both parameters), and rand() is a random number uniformly distributed in interval (0,1). The parameter *w* is a number, called the inertial weight, in the range [0,1], which specifies the weight by which the particle current velocity depends on its previous velocity and how far the particle is from its iteration best and global best positions. Our Numerical experiments have shown that the PSO algorithm converges fasterif *w* is linearly damped with iterations starting at 0.9 and decreasing linearly to 0.4 at the last iteration.

4- Move the particle. Once the velocity has been determined, it is simple to move the particle to its next location. The new coordinate is computed for each of the dimensions according the following equation

(5)

5- Loop to step (2) until a criterion is met, usually a sufficiently good fitness or a maximum number of iterations.

These steps are summarized as a pseudo code as illustrated in figure1 and was implemented with matlab.

Step (1):

Initialize particles with random positions and velocities in *M* dimensions.

Step (2):

For each particle, evaluate the optimization fitness function in *M* variables.

Step (3):

Update the particle velocity according to the following equation:

Step (4):

Update position according to the following equation:

Step (5):

Go to Step (2) until a stopping criterion is met, usually a predefined number of iterations.

Fig**.**  7: PSO Pseudo Code

**Figure 3.** Flowchart of the Proposed PSO algorithm

Yes

Yes

No

No

Initialize population with random position (x) and velocity (v) vectors

For each time step

For each particle

Evaluate fitness

If fitness(x) < fitness(gbest)

If fitness(x) < fitness(iterbest)

Update velocity

Update position

gbest= parameters of the best solution

gbest=x

iterbest=x

**4.3 Parallel PSO**

**CUDA Implementation:**

There are two parallel variants using CUDA/GPU for PSO algorithm [5] or for any evolution algorithm one global and other local or as paper [2] called embedded.

Concerning the global approach, Global variant was programmed on the GPU:

**1‐ Global:** Where all the mathematical calculations are parallelized, computing all fitness function, velocity, and position which Executes Fitness Evaluation and Position and Velocity Update for all particles in parallel using two different kernels on GPU.

**2- Local:** Where the Whole entire algorithm executed on the Local Memory of GPU except the initialization of the particles.

We implement the global one due to our limited memory of our GPU and to be able to use as many particles as we need without limitation of memory.

The sequential PSO Algorithm was implemented as reference, to Compare the performance of parallel variants to it. In all parallel implementations, the programming strategy involved the creation of one thread for each PSO particle. The rule was to replace all the sequential loops (specifically those where the iterations were in terms of the particles number) by a single multithreading kernel call. Thus, the sequential PSO algorithm and its parallel implementations have essentially the same structure. In the loops of the sequential code, each loop’s iteration is independent from all others [5].

CUDA programs launch parallel kernels with the following function‐call syntax:

Kernel<<<dimGrid,dimBlock>>>(…parameter\_list…);

Where dimGrid and dimBlock are specialized parameters that specify the dimensions of the parallel processing grid of blocks and the dimensions of the blocks of threads, respectively. Figure 3 depicts the structure of the sequential PSO algorithm where the following functional blocks can be observed [3]:

‐ **Population initialization**. It initializes each particle of the population in a random form.

‐ **Fitness function evaluation**.

‐ **Comparison**. It determines if an individual has better fitness than the best registered.

**‐ Imitation (updating)**. Every individual updates its position influenced by its own experience, and by the social environment.

The main idea is to create one thread for each PSO particle, as we will see below. Note that in the sequential PSO version (see Figure 3) all the functional modules are executed on the host processor. In the first parallel variant, the Global one, any arithmetic calculation is distributed to the GPU, replacing both the fitness function evaluation and position update modules by the associated kernel calls (see Figure 4) Use one thread for each particle in each kernel as we use 2 kernels: Kernel1 for (update the position of all particles) and Kernel2 for (evaluate the fitness of all particles).

In the second parallel variant, the Local one, only the initialization module remains running on the host processor, and there are a single kernel call associated to the evaluation, comparison and imitation modules, that run on the GPU until a termination condition is reached.

There are some practical considerations that must be taken into account to achieve a functional implementation of the parallel PSO algorithm on a GPU as stated in [5]:

**Overhead**. The GPU presents an overhead due to memory transferences between the host and the GPU device which are necessary during the information exchange. Because these transferences are relatively slow, any parallel implementation on a GPU must minimize their employment. Considering the overhead, it is understandable that the global variant (Global) are slower than the Local one, due to the information exchange between the host and the GPU during the algorithm execution. In the global variants, the information exchange is necessary since the host processor needs to know the information originated at both the evaluation and update modules in order to take any decision [5].

**Synchronization**. Before any decision branch, for example during the comparison process, all the running threads must be synchronized at the points where it is necessary to obtain unambiguous information. This point is particularly important when the threads have to communicate among themselves to share information [5].

Pseudo code:

*<initialize positions/velocities of all particles>*

*<perform a first evaluation of the fitness functions> kernel2*

*<set initial personal/iteration/global bests>*

**for(i = 0; i < Iterations\_Number; i++)**

**{**

*<update the position of all particles> kernel1*

*<re−evaluate the fitness of all particles> kernel2*

**}**

*<retrieve global best information to be returned as final result>*

Fig 2: Pseudo Code of the Parallel PSO algorithm using CUDA

**Open MP Implementation:**

PSO is easy to implement, and relatively easy to make parallel. PSO is especially easy to make parallel with Open-MP. Consider the main loop for PSO. The following pseudo code shows what is performed for each particles of PSO.

For each particle

**{**

*update the position of particle*

*re−evaluate the fitness of particle*

**}**

Notice the loop that begins with “For each particle”. This loop can be made parallel. In this loop each particle must update its position and evaluated against the new position. This takes considerable time, especially if the particle’s dimension is large.

Each particle is calculated independent of the other particles. The only overlap between particles is to determine the best particle. However, the best particle is determined after this loop is done. Because of this each particle can be evaluated in parallel, while the other particles are being evaluated.

When using Open-MP, very few changes are needed to make this change. If I were working with a lower level multithreading method, I would have to write considerably more plumbing/synchronization code. With Open-MP, the amount of code I must write is minimal. We will now modify the above code to make use of multi-threading. The modified code is shown here.

#pragma omp parallel for

For each particle

**{**

*update the position of all particles*

*re−evaluate the fitness of all particles*

**}**

The change is only one line. for anyone who has ever struggled with threads, making a program multi-threaded with a single line change is amazing! Typically you must write many lines of code, implementing thread pools. Dealing with critical sections, semaphores, and mutexes.

The only line of code that changed is the **pragma** statement just before the **for loop**. This instructs Open-MP to run each iteration of the for loop in parallel.

4.4 PSO Experimental Result

**PSO CUDA Results:**

Here we propose the first version of the adaptive beam forming application with PSO using CUDA and this is sample of the experimental results that show CPU time and GPU time of PSO for five different test cases. Experiments were run on a PC equipped with an Intel Core(TM)2Duo processor running at 2.80 GHz with a NVIDA GeForce 9600GT video card from NVIDIA corporation. All of the simulation runs were performed under the following settings:

Number of antenna in antenna array = 24

Number of Particle = 40

Number of Iterations = 150

The sequential execution of the program took 4446 ms while running the PSO algorithm on GPU NVIDA (GeForce 9600) the execution time was only 265 ms. In particular the achieved running speedup was of about 16.5 times as illustrated in figure 5.In Table 2, the first column displays images illustrating the optimum normalized radiation pattern resulted from the proposed PSO the second column shows figures that illustrate the optimum normalized radiation pattern measured in dB, and the last column illustrates the change of fitness value with iterations. The results were recorded for five different test cases[4].

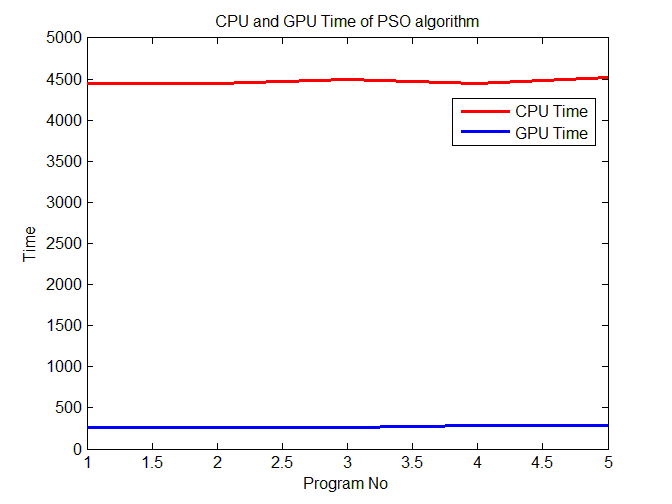


Figure 4 : sample runs and average speed up =16.5

**PSO Open-MP Results:**

We will now present the performance of the parallel version of PSO using open-mp compared to the single-threaded version. The tests were performed on an i5 Intel Core. The following stats were used:

Number of antenna in antenna array = 24

Number of Particle = 180

Number of Iterations = 500

 The following results were observed.

|  |  |
| --- | --- |
| **PSO** | |
| **Sequential** | **Parallel** |
| Time Elapsed in update position and Acceleration and fitness for all props in seconds = 0.010000 sec | Time Elapsed in update position and Acceleration and fitness for all props in seconds = 0.004000 sec |
| Time Elapsed in initialize props in seconds = 0.000000 sec | Time Elapsed in initialize props in seconds = 0.000000 sec |
| Time Elapsed in calculate fitness for all props in seconds = 0.000000 sec | Time Elapsed in calculate fitness for all props in seconds = 0.000000 sec |
| Time Elapsed in get best fitness for all props in seconds = 0.000000 sec | Time Elapsed in get best fitness for all props in seconds = 0.000000 sec |
| Total Time Elapsed = 5.000000000000000000000 sec | Total Time Elapsed = 2.000000000000000000000 sec |

**Table 5:** Results of sequential and parallel PSO

As seen above in table 5, multi-threading gave a large performance using 16 threads.

Figure 8. .Flow Chart of the Parallel PSO algorithm

Update personal / iteration / global bests

Yes

Initialize PSO Parameter

Initialize Position /Velocity of all Particles

Fitness Evaluation for Particle (1)

Fitness Evaluation for Particle (2)

Fitness Evaluation for Particle (N)

Set initial personal / iteration / global bests

I < iteration\_num

Update Position for Particle (1)

Update Position for Particle (2)

Update Position for Particle (N)

Fitness Evaluation for Particle (1)

Fitness Evaluation for Particle (2)

Fitness Evaluation for Particle (N)

………………………………………………….

………………………………………………….

Retrieve global best information

No

………………………………………………….

|  |  |  |  |
| --- | --- | --- | --- |
|  | **(a) Radiation Pattern** | **(b)y-axis is Radiation Pattern in dB and x-axis is angle in radian** | **(c) y-axis is fitness value and x-axis is iteration number** |
| Desired:  0, 90, 180  Undesired  30 | 0-90-180-30-rad.bmp | 0-90-180-30-log.bmp | 0-90-180-30-iter.bmp |
| Desired:  0, 60  Undesired  :180,30 | 0,60,180,30-rad.bmp | 0,60,180,30-log.bmp | 0,60,180,30-iter.bmp |
| Desired:  180, 60  Undesired  240, 30 | 180-60-240-30-rad.bmp | 180-60-240-30-log.bmp | 180-60-240-30-iter.bmp |
| Desired:  180  Undesired  60, 240 | 180-60-240-rad.bmp | 180-60-240-log.bmp | 180-60-240-iter.bmp |
| Desired:  180  Undesired  :300, 120, 30 | 180-300-120-30-rad.bmp | 180-300-120-30-log.bmp | 180-300-120-30-iter.bmp |

Table 1 : Experimental results of PSO using CUDA

|  |  |  |  |
| --- | --- | --- | --- |
|  | **(a) Radiation Pattern** | **(b)y-axis is Radiation Pattern in dB and x-axis is angle in radian** | **(c) y-axis is fitness value and x-axis is iteration number** |
| Desired:  0, 90, 180  Undesired  30 | 0-90-180--30 rad.bmp | 0-90-180--30 rad2.bmp | 0-90-180--30 iter.bmp |
| Desired:  0, 60  Undesired  :180,30 | 0-60--180--30 rad.bmp | 0-60--180--30 rad2.bmp | 0-60--180--30 iter.bmp |
| Desired:  180, 60  Undesired  240, 30 | 180-60--240--30  rad.bmp | 180-60--240--30  rad2.bmp | 180-60--240--30  iter.bmp |
| Desired:  180  Undesired  60, 240 | 180--60--240  rad.bmp | 180--60--240  rad2.bmp | 180--60--240  iter.bmp |
| Desired:  180  Undesired  :300, 120, 30 | 180--300--120--30 rad.bmp | 180--300--120--30 rad2.bmp | 180--300--120--30 iter.bmp |

Table 1 : Experimental results of PSO using open-mp