INTERNATIONAL JOURNAL OF COMMUNICATION SYSTEMS
Int. J. Commun. Syst. 2013; **26**:888–911
Published online 22 November 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/dac.1376

Distributed topology control in large-scale hybrid RF/FSO networks: SIMT GPU-based particle swarm optimization approach

Osama Awwad[1], Ala Al-Fuqaha[1,*,†], Ghassen Ben Brahim[2], Bilal Khan[3] and Ammar Rayes[4]

[1]*Computer Science Department, Western Michigan University, Kalamazoo, MI 49008 USA*
[2]*Integrated Defense Systems, Boeing Company, Huntington Beach, CA 92647 USA*
[3]*John Jay College, City University of New York, New York, NY 10019 USA*
[4]*Advanced Support Systems, Cisco Systems, San Jose, CA 95134 USA*

SUMMARY

The tremendous power of graphics processing unit (GPU) computing relative to prior CPU-only architectures presents new opportunities for efficient solutions of previously intractable large-scale optimization problems. Although most previous work in this field focused on scientific applications in the areas of medicine and physics, here we present a Compute Unified Device Architecture-based (CUDA) GPU solution to solve the topology control problem in hybrid radio frequency and free space optics wireless mesh networks by adapting and adjusting the transmission power and the beam-width of individual nodes according to QoS requirements. Our approach is based on a stochastic global optimization technique inspired by the social behavior of flocking birds — so-called 'particle swarm optimization' — and was implemented on the NVIDIA GeForce GTX 285 GPU. The implementation achieved a performance speedup factor of 392 over a CPU-only implementation. Several innovations in the memory/execution structure in our approach enabled us to surpass all prior known particle swarm optimization GPU implementations. Our results provide a promising indication of the viability of GPU-based approaches towards the solution of large-scale optimization problems such as those found in radio frequency and free space optics wireless mesh network design. Copyright © 2011 John Wiley & Sons, Ltd.

Received 9 February 2010; Revised 29 September 2011; Accepted 30 September 2011

KEY WORDS: CUDA; GPU; hybrid RF/FSO; PSO; QoS; topology control; wireless mesh networks

1. INTRODUCTION

Adjusting the beam-width and the transmission power in hybrid radio frequency and free space optics (RF/FSO) mesh networks presents a competing local versus global trade-off. A node with a large beam width or high transmission power usually has more nodes in the transmission range and hence a higher degree, which reduces the average global path length, thus minimizing the end-to-end delay of multihop connections. However, a higher node degree implies higher link layer interference, which reduces local throughput because of high channel contention. On the other hand, nodes that employ a narrow beam width or low transmission power have a lower number of nodes in their transmission range, hence a lower node degree — this results in higher average global path lengths and in turn, high end-to-end delay. However, a lower node degree implies lower interference, which tends to ameliorate local throughput because channel contention decreases. As such, it appears that there is a trade-off and our objective is to construct a robust topology by minimizing the transmission power, adapting the beam width, and selecting different channels such that we meet joint throughput and end-to-end delay requirements.

*Correspondence to: Ala Al-Fuqaha, Computer Science Department, Western Michigan University, Kalamazoo, MI 49008 USA.
†E-mail: alfuqaha@cs.wmich.edu

In this work, we introduce our new approach to solve the topology control problem in large-scale hybrid RF/FSO networks (where by large we mean in terms of number of nodes, number of transceivers, and number of source–destination pairs). Our approach is based on the 'particle swarm optimization' (PSO) technique that was first introduced in 1995 by Kennedy and Eberhart [1]. PSO shares many similarities with the genetic algorithms techniques [2, 3] in the sense that a system is initialized with a population of random solutions and then searches for a better solution by updating its population over generations. However, PSO has no operators such as crossover and mutation. In contrast, the potential solutions in PSO — called 'particles' — fly through the problem space by following the current optimum particles.

Particle swarm optimization is a metaheuristic algorithm that provides a general purpose optimization technique in which there are few parameters to adjust. Because of this, the same concrete instantiation of PSO appears to work well in a wide number of problem instances and in varying application contexts. The large number of particles moving in the solution space prevents the PSO algorithm from becoming trapped in local optima. Because PSO is by nature a decentralized computation, in which particles interact and exchange information amongst each other without requiring centralized coordination, the technique extends readily to distributed environments. Although there have been prior efforts to consider graphical processing unit (GPU)-based PSO [4–6], this paper presents results of what are, to our knowledge, the first ever attempt to use PSO techniques in the area of hybrid RF/FSO wireless networks.

The rest of this paper is organized as follows. In Section 2, we introduce GPU computing using the Compute Unified Device Architecture (CUDA) platform. In Section 3, we design the PSO structure for topology control problem in hybrid RF/FSO wireless networks. In Section 4, we introduce a parallel PSO algorithm using the CUDA platform. In Section 5, we describe our experimental results and interpret the observed outcomes. Finally, in Section 6, we present overall conclusions and the future trajectory of our research efforts.

2. GRAPHICAL PROCESSING UNIT OVERVIEW

Pure CPU architectures are facing an apparent scalability barrier: performance gains are no longer commensurate with the increases in the number of transistors. In the meantime, GPU manufacturers such as NVIDIA and ATI have started to take advantage of their graphics architectures to provide tremendous computational power. For example, in 2008, NVIDIA's GeForce GTX 280 was able to execute around 1000 GFLOPS compared with 100 GFLOPS by the Intel Harpertown Quad-Core 3.2 GHz CPU. The comparison of CPU and GPU performance benchmarks is depicted in Figure 1.

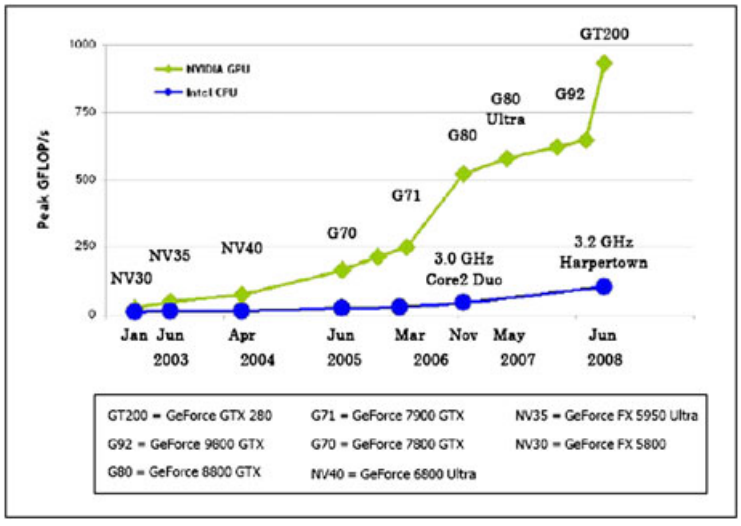

Figure 1. Floating point operations for CPU and GPU [11].

DOI: 10.1002/dac

To enable users to harness the processing power of a GPU to solve general-purpose computing problems, NVIDIA introduced a high-level programming model called CUDA in 2007. CUDA provides a C-like language with custom NVIDIA extensions for handling parallelism. Together with the GPU card's massive parallelism, high memory bandwidth, and affordability,[‡] CUDA makes GPU computing an attractive solution for many scientific applications and large-scale computational problems. For example, Preis *et al*. developed a GPU-accelerated Monte Carlo simulation [7] with which they achieved a 60× speedup over their earlier CPU-only implementation. Januszewski and Kostur were able to achieve a speedup of 675× in computing numerical solutions of stochastic differential equations using CUDA [8]. In addition, GPU solutions have been developed for many medical applications, particularly magnetic resonance imaging [9, 10]. For example, some CUDA-enabled GPU-based software developed recently [9] allows a neurosurgeon to visualize the neuronal fibers in the brain of a patient.

2.1. Compute unified device architecture hierarchal architecture

Graphical processing unit computing is considered a single instruction multiple thread (SIMT) model, targeting problems that can be solved by executing in parallel threads on many data elements. To move beyond nongraphical applications and into general purpose parallel programming, NVIDIA introduced the CUDA model, which enables programmers to write their own code using a standard programming language like C with NVIDIA extensions. CUDA provides a scalable model whereby the program can be executed on any number of cores, and only the runtime system needs to know the physical processor count.

The CUDA organizes parallelism in a hierarchal system, which is composed of three levels: grid, block, and thread. At the top of the hierarchy, a grid of multiple thread blocks will be created when the CPU host invokes a GPU device function called 'kernel'. This causes all blocks defined in that grid to be distributed to available multiprocessors. Each block contains multiple threads. Because each GPU thread lies in a block that lies in a grid, all threads must execute the same kernel. The CUDA runtime system assigns a unique ID composed of `blockId` and `threadId` to distinguish between threads. At present, CUDA only allows a single grid to run at a time, meaning that different kernels cannot run simultaneously. The CUDA syntax to invoke a kernel is simply

```
kernelname<<<dimGrid, dimBlock>>>(arg1, arg2, ...etc)
```

For example, in Figure 2, '*vecAdd*' is a kernel defined in a grid that contains two blocks and each block contains three threads. All six threads will execute the *vecAdd* kernel simultaneously.

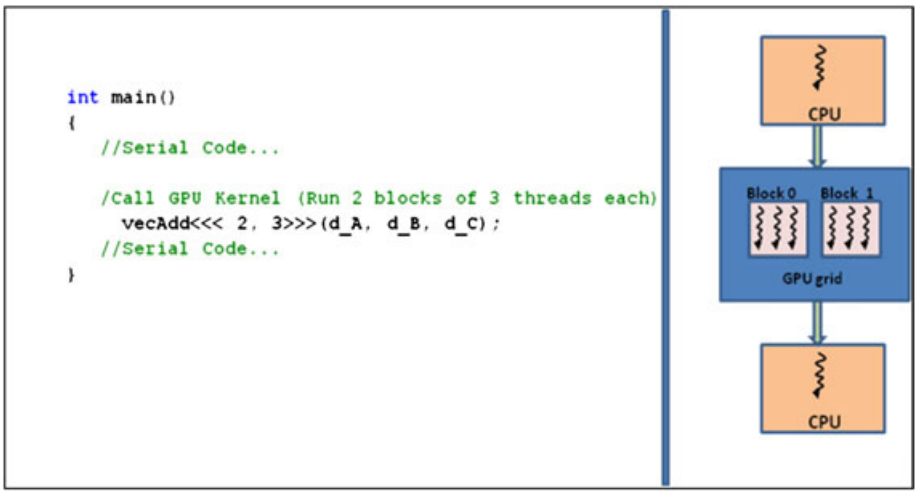

Figure 2. Kernel execution by six threads over two blocks.

[‡]At the time of printing, the GeForce GTX 280 with 240 processors cost less than $400.

DOI: 10.1002/dac

2.2. *Compute unified device architecture memory access*

During kernel execution, threads have access to different types of GPU memories.

- Global memory: Device memory, Read/Write (R/W) per-grid
- Constant memory: Cached device memory, Read Only (RO) per-grid
- Texture memory: Cached device memory, RO per-grid
- Local memory: Device memory, R/W per-thread
- Shared memory: On-chip memory, R/W per-block

In addition to the above register, memory provides limited on-chip storage that supports fast R/W access per-thread in one clock cycle. The types of memory are illustrated in Figure 3.

Shared memory and registers are the fastest and costliest — and so are also the most limited in size. At the other extreme, device memories are large but must be accessed with high latency and low bandwidth compared with on-chip memories. GPU architectures allow the latency of device memory (and particularly 'Global memory') to be somewhat masked because the multiprocessor executes threads in 32 parallel thread groups called warps. This feature — exported into the CUDA API — allows the amortized bandwidth of global memory access to be maximized by performing simultaneous half-warp accesses of coalesced global memory using a single memory transaction of 32, 64, or 128 bytes.

2.3. *Programming in compute unified device architecture*

The general processing flow of every CUDA program includes the following five steps:

(1) Allocate memory on the GPU device.
(2) Copy data from system memory into GPU memory.
(3) Call the kernel function from CPU host. Whenever a kernel is invoked it will be running N times in parallel in N separate threads on a CUDA-enabled device.

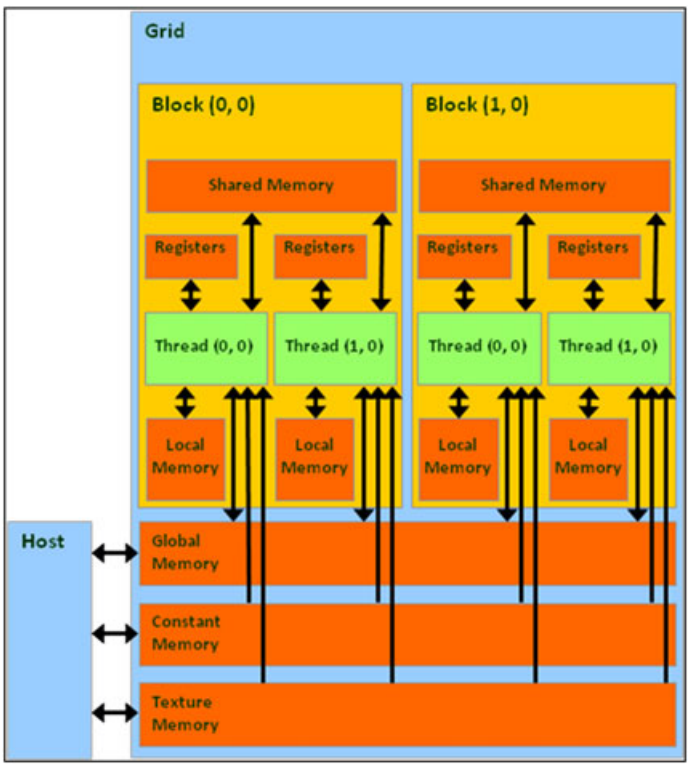

Figure 3. Memory types available in GPU [11].

DOI: 10.1002/dac

(4) GPU device executes instructions using all available cores in parallel.
(5) Copy the output results back from GPU memory into host system memory.

To illustrate this, let us consider the vector addition example [11] in Figure 4. The program starts by allocating and copying *d_A*, *d_B*, and *d_C* vector into the GPU global memory. Next, the kernel will be executed by each thread in the grid; each thread performs one pairwise addition. Note that the grid contains six threads which are distributed into two sets of three threads — one set is assigned to each of the two blocks in the grid. The last step (5) of the program is to copy the results from the device global memory into the host memory.

3. PARTICLE SWARM OPTIMIZATION

In this section, we will describe how we adapt the PSO technique to solve the topology control optimization problem for hybrid RF/FSO in wireless mesh networks. These details include: (i) how the topology control solution of hybrid RF/FSO is encoded in each particle; (ii) how initial particles are generated in the swarm; (iii) how particles move in the search space to reach a better solution; and (iv) how the fitness function is evaluated.

3.1. Particle swarm optimization overview

Swarm intelligence falls in the category of artificial intelligence techniques based on the collective behavior of decentralized, self-organizing systems. These systems are modeled by a population of agents that share information with each other and interact with their environment. Although there is no centralized mechanism to govern agent interactions, a random interaction between agents leads to global systemic intelligence. Examples of such systems in nature include ant colonies, flocks of birds, schools of fish, etc. Many algorithms have been developed that adopt this vantage point in their

```
int main()
{
   //Serial Code...
   // allocate and initialize host (CPU) memory
   float *h_A = …, *h_B = …;

   // 1) allocate device (GPU) memory
   float *d_A, *d_B, *d_C;
   cudaMalloc( (void**) &d_A, N * sizeof(float));
   cudaMalloc( (void**) &d_B, N * sizeof(float));
   cudaMalloc( (void**) &d_C, N * sizeof(float));

   // 2) copy host memory to device
   cudaMemcpy( d_A, h_A, N * sizeof(float),
   cudaMemcpyHostToDevice) );
   cudaMemcpy( d_B, h_B, N * sizeof(float),
   cudaMemcpyHostToDevice) );
   //3) Call GPU Kernel (Run 2 blcksw 3 thrds each)
   vecAdd<<< 2, 3>>>(d_A, d_B, d_C);

   //5) copy device memory output to host
   cudaMemcpy( h_C, d_C, N * sizeof(float),
   cudaMemcpyDeviceToHost) );

}

//4) Kernel Definition
__global__ void vecAdd(float* A, float* B, float* C)
{
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   C[i] = A[i] + B[i];
}
```

Figure 4. Vector addition example using CUDA.

DOI: 10.1002/dac

strategy. The control of unmanned vehicles in the US army, the planetary mapping in NASA, crowd simulation in movies or games, and ant-based routing are just a few examples of recent applications [2,12,13]. PSO is considered a subfield of swarm intelligence and is most suited for problems where the solution can be represented by a point on a smooth surface in an n-dimensional space.

An optimization problem is modeled within the PSO framework by constructing a population (called the swarm) of candidate solutions (called particles) that collectively evolve toward better solutions by exchanging information between themselves. The evolution of the swarm in PSO is an iterative process whereby each particle adjusts its solution in the direction of its own best historical solution, while at the same time moving towards the direction of the best solution obtained by other particles in the swarm. To achieve this, each particle keeps track of two best values: (i) its best fitness value, which is stored as the best local value, and (ii) the best global fitness, which is computed based on the best local values of all the particles in the swarm. Some PSO variants also keep track of the best value obtained so far by particles in the vicinity of each particle. The movement of the particle in the search space is controlled at each iteration step by the particle position ($x_{i,j}$) and velocity ($v_{i,j}$) update equations

$$v_{i,j} = wv_i + c_1 r_1(globalbest_j - x_{i,j}) + c_2 r_2(localbesti_{ij} - x_{i,j})$$
$$x_{i,j} = x_{i,j} + v_{i,j}$$

where i is the particle ID, j is the variable ID, w is the inertia weight, $c1$ and $c2$ are two positive constants, called the *cognitive* and *social* parameters, respectively, and $r1$ and $r2$ are random numbers uniformly distributed in [0, 1].

3.2. *Hybrid radio frequency and free space optics system model*

We consider a hybrid RF/FSO wireless ad hoc network consisting of N nodes equipped with directional FSO transceivers and omnidirectional RF transceivers having limited transmission range. We assume the light source for FSO channels is noncoherent light emitting diodes and that each FSO transceiver carries data on its own unique wavelength. Hence, multiple FSO transceivers can send and receive data at the same time without interfering with each other.

The RF and FSO propagations suffer from different losses and attenuations [14]. In our study, we focus on the attenuations that occur from geometrical loss. If node i transmits with power P_t the power of the signal received by node j, P_r, is given by [14]

$$P_r = P_t \left(\frac{D_t}{D_t + 100d\theta} \right)^2,$$

where D_t is the transceiver diameter, d is the distance between node i and node j, and θ is the beam divergence angle. Thus, each wireless channel has a computable BER that is the probability of the occurrence of an error during data transfer over that link. The relationship between the BER of a wireless channel and the received power level P_r is a function of the modulation scheme. Within the RF channel, we will consider the instantaneous channel BER that is given in [15] based on a noncoherent binary orthogonal frequency shift keying modulation scheme

$$BER = \frac{1}{2} erfc \left(\frac{-P_r}{2P_{\text{noise}}} \right),$$

where P_{noise} is the RF noise power. In the FSO channel, we will consider BER that is given in [16] based on the on–off keying modulation scheme

$$BER = \frac{1}{2} erfc \left(\frac{R.P_r}{2\sqrt{2P_{\text{noise}}}} \right),$$

where R is the photodetector responsiveness and P_{noise} is the FSO noise power. For more details we refer the reader to our work in [17].

DOI: 10.1002/dac

3.3. Particle encoding

A particle is encoded as a vector of heterogeneous variables whose values, taken together, represent a candidate solution of the hybrid RF/FSO topology control problem. The constituent variables can be classified into three types:

- **X_Power(i,j,t)**: Represents the transmission power from node i to node j using transceiver t. $\forall i, j \epsilon V$ and t ϵ T
- **X_Beam(i,j,t)**: Represents the beam opening from node i to node j using transceiver t. $\forall i, j \epsilon V$ and t ϵ T
- **X_SD(sd):** Represents a pointer to entries in the routing and transceiver assignment enumeration table $\forall sd \epsilon SD$ which is carried out by taking all the routes generated by the K-shortest paths algorithm for each source–destination pair. After all routes are generated, all possible transceiver assignment combinations are generated. Each unique transceiver assignment on a route is considered as a unique path. This table is generated by combining the K-shortest routes for each source–destination pair with all the possible transceiver assignments for that route.

Figure 5 shows the structure of the variables that comprise each particle. To help make our encoding of the X_SD variables more concrete, consider the example depicted in Figure 6, which represents a simple three-node network. Here, a single X_SD variable points to a candidate route and transceiver assignment. As seen in the figure, the variable points to index 4 entry of the routing and transceiver assignment enumeration table. Notice that the entries of the enumeration table have full routing and transceiver assignment information for each connection. For example, the enumeration matrix indicates that the index 4 entry uses transceiver 2 on the first hop and transceiver 1 on the

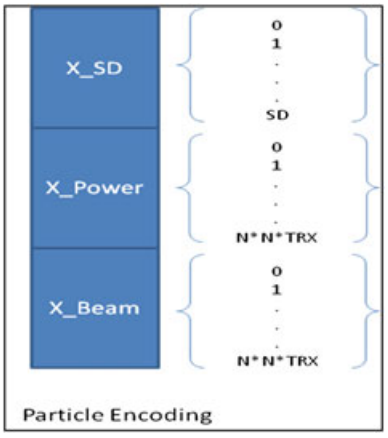

Figure 5. Particle encoding of a solution to the topology control problem in the hybrid RF/FSO networks.

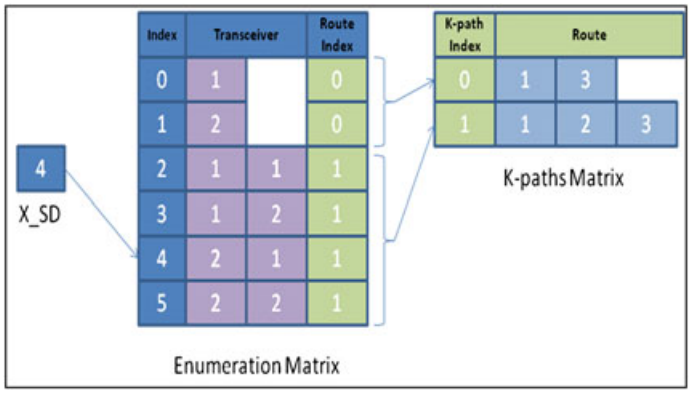

Figure 6. X_SD variable encoding example using enumeration matrix and K-paths matrix.

DOI: 10.1002/dac

second hop. Also, the route index field indicates that the first hop for this entry is (1➔2) and the second hop is (2➔3) by pointing to the K-path matrix.

3.4. Swarm initialization process

In our experiments, rather than randomly generating the initial state of all particles, we initialized some of the particles using the first-fit algorithm [18]. The first-fit procedure searches the routing and transceiver assignment enumeration table from the beginning and uses the first possible path that meets the QoS requirements to satisfy the request. If no free entry is available, a random path is selected and the request is marked as unsatisfied. To reduce the possibility of being trapped in local optima, we mixed particles that were initialized via first-fit with other particles that were initialized with random values. The flowchart in Figure 7 illustrates the swarm initialization process. The new generating process and its variable initialization is detailed in the pseudocode in Figure 8. Note that the source–destination pairs are sorted according to their end-to-end delay requirements, and throughput requirements are used to break ties.

3.5. Particle update process

Particles in PSO move in an attempt to improve their solutions according to the particle velocity and position equations

$$V_{i,j}^{k+1} = wV_{i,j}^{k} + c1r1_{i,j}^{k}(P_{i,j}^{k} - X_{i,j}^{k}) + c2r2_{i,j}^{k}(g_{j}^{k} - X_{i}k)$$

$$X_{i,j}^{k+1} = X_{i,j}^{k} + V_{i,j}^{k+1}$$

where:

- $i = 1, \ldots, N$ is the particle, N is the population size
- $j = 1, \ldots, M$ is the variable, M is the number of variables
- k is the iteration number

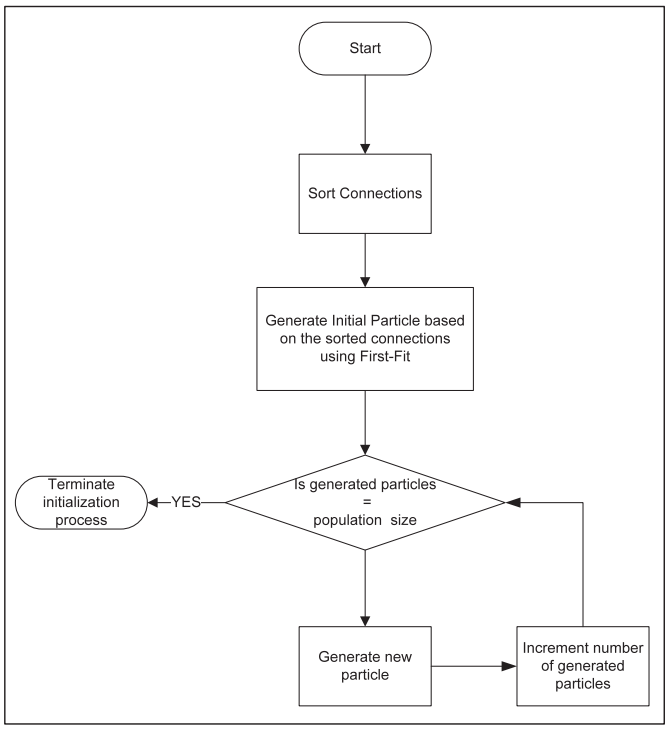

Figure 7. Swarm initialization.

DOI: 10.1002/dac

```
generateNewParticle: Function generates new particle
SD: Number of source-destnation pairs
R: Integer value to indicate the percentage of
   random particles in the initial swarm
ParticleID: The Id of the new particle
generateIntRandom(N): Function generates int [0, N]
assignRandomPath: Function assigns a random path
index for the source-destination connection
runFirstFit: Function assigns a path index for the
source-destination using first-fit algorithm
Output: Particle

Particle generateNewParticle(particleId)
{
   if(ParticleID==0)
      for(int sd=0;sd<SD;sd++)
      {
       Particle[i]=runFirstFit(sd);
      }
   else
   {
     int sdCounter=0;
     boolean selectedConnection[SD];
     for(int sd=0;sd<SD;sd++)
      selectedConnection[i]=false;
     while(sdCounter!=SD)
     {
        sd=generateIntRandom(SD);
        if(selectedConnection[sd]==-1)
        {
          if(particleID%R==0)
           Particle=assignRandomPath(sd,ParticleID);
          else
           Particle=runFirstFit(sd,ParticleID);
          selectedConnection[sd]=1;
          sdCounter++;
        }
      }
    }
 return Particle
}
```

Figure 8. Generating new particle algorithm.

- $P_{i,j}^k$ is the value of variable j in the optimal position of particle i seen so far.
- g_j^k is the value of variable j in the optimal position seen so far (over all particles).
- w is the inertia weight
- $c1$ and $c2$ are two positive constants, called the cognitive and social parameter, respectively

$r1_{i,j}^k$ and $r2_{i,j}^k$ are random numbers uniformly distributed within the range [0, 1]. In the velocity equation, the inertia weight w controls the effect of previous velocities of the particle in its current velocity. Large values of w facilitate the 'exploration ability' of the particle, whereas small values favor exploitation of the local search area. Experimental results show that it is preferable to set w to a large value at the beginning of the search and then gradually decrease it when the particle approaches the best fitness value [19, 20].

The relative magnitudes of $r1_{i,j}^k \times c1$ and $r2_{i,j}^k \times c2$ determine whether the particle moves towards pBest or gBest. If the upper bound of $r1_{i,j}^k \times c1$ is greater than the upper bound of $r2_{i,j}^k \times c2$, then the particle tends to utilize the neighborhood experience more than its own experience toward finding a better fitness value. The values $c1$ and $c2$ are generated randomly for each particle at each iteration so that the particles may vary the influence between different sources of information [19].

In our PSO model, we bounded the velocity of each variable in a range $[V\min, V\max]$ to prevent numerical errors from accumulating and to prevent the swarm from exploding or coming to a standstill. Because the variables in our topology control problem are heterogeneous, we used a unique

DOI: 10.1002/dac

array data structure for every variable type and by doing that we can define different velocity intervals for each variable type. Figure 9 provides the C structures of our swarm implementation. As can be seen, the swarm data structure is application-independent and PSO variables can be customized using the same structure based on the specific application involved.

For example, to add a variable called '***X_z***', a *struct* for that variable needs to be defined as follows:

```
typedef struct
{
  int* X_z;
}ZVAR;
```

Then a pointer to that variable inside the swarm should be defined as follows:

```
ZVAR ** zVar;
```

3.6. *Optimization problem and fitness function evaluation*

To begin the process of constructing a fitness function, we start by modeling our topology control optimization problem as a nonlinear programming problem. Our objective is to construct a robust topology by minimizing the transmission power, adapting the beam width, and selecting different channels such that we meet joint throughput and end-to-end delay requirements.

```
typedef struct
{
        int * X_sd;
}SDVAR;

typedef struct
{
        float * X_Power;
}PVAR;

typedef struct
{
        float * X_Beam;
}TVAR;

typedef struct
{
          SDVAR ** sdVar;
          PVAR **  pVar;
          TVAR **  tVar;

          SDVAR ** V_sdVar;
          PVAR **  V_pVar;
          TVAR **  V_tVar;

          SDVAR ** pl_sdVar;
          PVAR **  pl_pVar;
          TVAR **  pl_tVar;

          int * pg_sdVar;
          float * pg_pVar;
          float * pg_tVar;

          float * plFitness;
          SDVAR ** sdVarOrder;

          float  bestSwarmFitness;
          int numBlocking;

}Swarm;
```

Figure 9. C structures of our swarm implementation.

DOI: 10.1002/dac

Assumptions

- The network topology is a mesh with directed links.
- At any given node, we have RF and FSO transceivers.
- RF transceivers are omnidirectional, while FSO transceivers are directional.

Input

- V: Set of mobile nodes. For each node $i \epsilon V$, we have:
  - Location
  - Number of RF and FSO transceivers
- T_i: Set of transceivers. For each transceiver t at node i,we have:
  - C_MAX: Maximum capacity
  - S: Sensitivity
  - D: Diameter
  - W_{max}: Max Beam-Width
  - P_{max}: Max Power level
- SD: Set of requested source–destination connections. For each $(s, d)\epsilon\ SD$, we have:
  - $D_{(s,d)}$: Maximum delay
  - $Th_{(s,d)}$: Minimum Throughput
- Paths: The enumeration look-up table of all possible paths, which is carried out by taking all the routes generated by the K-shortest paths algorithm for each source–destination pair. After all routes are generated, all possible transceiver assignment combinations are generated. Each unique transceiver assignment on a route is considered as a unique path.

Variables

For the purpose of writing the equations precisely, we will rename some of the PSO variables now, as follows:

- H_{sd}: Number of hops in the selected path in *X_sd[sd]*.
- X :$[x_{i,j,c}]$: Transmission power (*X_Power*) for every node i and j and transceiver c in the network.
- A :$[a_{i,j,c}]$: Beam opening (*X_Beam*) for every node i and j and transceiver c in the network.
- G :$[g_{i,j,c}]$: Link incidence matrix of size $NxNxT$ where N is the number of nodes and T is the number of transceivers. $g_{i,j,c}$ stores the reserved capacity of every link ($i \rightarrow j$ using transceiver c). The value of $g_{i,j,c}$ gets updated for every used link in the selected path in *X_sd[sd]*.

3.6.1. Objective function. The objective function is to minimize the total transmission power in the network

$$Min \sum_{i-1}^{|V|} \sum_{j-1}^{|V|} \sum_{c-1}^{|T[i]|} x_{i,j,c} \tag{1}$$

Constraints

- Delay Constraints**:** To ensure that the number of hops in the selected route does not violate the delay requirement in terms of end-to-end delay

$$H_{sd} \leqslant D_{sd} \quad \forall \mathrm{sd}\ \epsilon SD$$

- Throughput Constraints: To ensure that the throughput requirements are met.
  For every link $(l_{i,j,c})$ in the selected path look-up table and

$$l_{i,j,c}.Th_{sd} \leqslant g_{i,j,c}.(1-BER_{i,j,c})$$

- Alignment Constraints: To ensure that every FSO transceiver in the network is aligned and transmitting with one transceiver only at the same time.

DOI: 10.1002/dac

For every selected link between node i and node j using transceiver c in the selected paths:

$$\begin{aligned} g_{i,j,c}.g_{i,k,c} &= 0 \quad \forall \mathrm{k} \neq \mathrm{j} \\ g_{i,j,c}.g_{k,i,c} &= 0 \quad \forall \mathrm{k} \neq \mathrm{j} \\ g_{i,j,c}.g_{j,k,c} &= 0 \quad \forall \mathrm{k} \neq \mathrm{i} \\ g_{i,j,c}.g_{k,j,c} &= 0 \quad \forall \mathrm{k} \neq \mathrm{i} \end{aligned}$$

- Power Constraints: To ensure that the transmission power on each transceiver does not exceed the maximum power

$$0 \leqslant x_{i,j,c} \leqslant P_{\max} \forall i,\ j \epsilon V \text{ and } \mathrm{c} \epsilon T_i$$

3.6.2. Fitness function evaluation. The constrained topology control problem above is transformed into an unconstrained problem and the fitness function Fx of our PSO model is calculated by adding a penalty component $\boldsymbol{\gamma(x)}$ for every unsatisfied constraint to the objective function value. Historical applications of PSO consider penalty functions of two kinds: stationary and nonstationary functions. In the former, fixed penalty is added to the value of the objective function when a constraint is violated. The latter approaches add a dynamically determined penalty value, which depends on how far the infeasible point is from the constraint. A previous work shows that the results obtained from the nonstationary approaches are more accurate than those obtained using stationary approaches [19].

By applying the nonstationary approach used in [19] to the penalty function, our fitness function is evaluated as follows:

$$Fx = F + h(k) \sum_{i}^{R} \gamma(i)$$

where R is the number of constraints. We use

$$h(k) = \sqrt{k}, \text{where } k \text{ is the iteration number.}$$

3.6.3. Sequential PSO algorithm. After defining the swarm structure and the fitness evaluation, the general algorithm of the PSO can be applied as presented in Figure 10. It starts by creating the swarm of particles and assigning each particle with its parameters such as the initial position. Then the algorithm updates the position of each particle according to the velocity/position equations.

```
Do
{
   For each particle
   {
     Calculate the corresponding fitness value
     If the fitness value is better than the
         particle's best fitness value then
          Set the current P vector to the particle's
           current X vector
   }
   Choose the particle with the lowest fitness value
and make it the global best position

   For each particle
   {
      Calculate the particle's velocity
      Update the particle current position vector  X
   }

} while maximum iteration or minimum error criteria
is not attained
```

Figure 10. Sequential PSO algorithm.

DOI: 10.1002/dac

In each iteration step, the particle compares its current position with its best position achieved to date. If the current position turns out to be better, the current position becomes the particle's best-ever position. The particle with the best value for the fitness function is chosen to be the swarm's best particle and the particles in the swarm tend to fly toward this particle.

During the update process in PSO, particles fly in the search space to improve the fitness of the global best solution. The direction and velocity of these particles are influenced by their parameter values $c1$, $c2$, and w. For example, in the velocity equation, having a large $c2$ compared with $c1$ can lead all of the swarm's particles to diverge to the direction of the current global best particle and not exploring the rest of the search space. On the other hand, having a $c1$ larger than $c2$ can lead the particles to diverge into their current local best particle. In addition, if the inertial velocity w is small, all particles might slow down until they approach zero velocity at the global best. Because 'large' and 'small' are fuzzy words that are application dependent in PSO, determining the values of these parameters is considered as a critical factor in PSO design to find a better solution. Moreover, having a general technique to adjust the PSO parameters can lead to one PSO version that works well in a wide variety of applications.

3.7. Particle swarm optimization kill process

In our experiments, we developed and implemented a 'kill process' to find accurate values for these parameters. The idea behind this approach is, we run multiple swarms simultaneously and we terminate the swarm that shows the least improvement. After that, the terminated swarm is initialized using its previous initial particles but this time using new constants. Figure 11 presents a flowchart of this technique.

4. PARALLEL SIMT PSO ALGORITHM

In this section, we introduce a novel parallel SIMT PSO solution to provide a large-scale solution with time efficiency for the topology control problem in a hybrid RF/FSO using CUDA-enabled

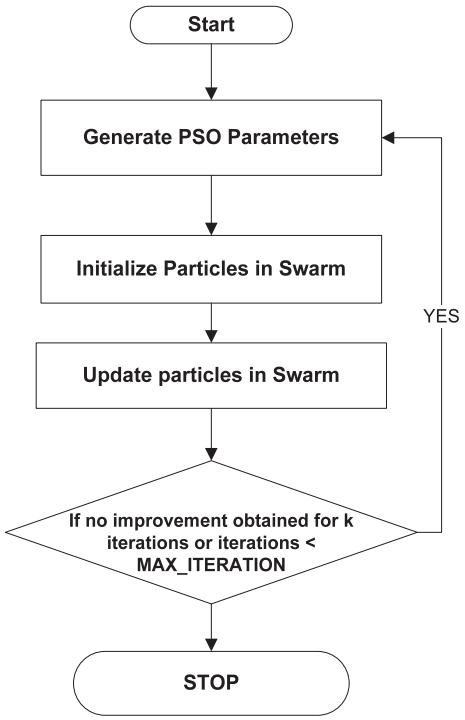

Figure 11. PSO kill process flow chart.

DOI: 10.1002/dac

GPUs based on the sequential version discussed in the previous section. In literature, a general PSO implementation in GPU has been provided in [4, 5].

Cai *et al*. developed a fine-grained parallel PSO [4]. Their implementation, however, was not efficient, as evidenced by the fact that the best speedup the authors were able to achieve was a factor of seven. Also, in some of the experiments, the authors found that the CPU implementation yielded higher speedup than the GPU-based one. The main reason for this was the authors' reliance on texture-rendering graphics, which is not as flexible as the CUDA-enabled GPU computing features developed subsequently by NVIDIA.

On the other hand, Zhu and Curry introduced the particle swarm with pattern search optimization algorithm using CUDA [5]. The authors provided a heterogeneous CPU–GPU solution, such that the heavy computation tasks were primarily executed on the GPU, and fitness evaluation was calculated using the CPU. The authors were able to achieve 200× speedup in some cases; however, even their PSO implementation cannot be considered an efficient implementation for large-scale problems for the following reasons:

(1) The extent of the GPU–CPU communication overhead in [5] was hidden because the size of the problem instances was quite modest. In their experiments, only 30 variables and 50 iteration steps were required. According to their parallel PSO algorithm, after every iteration step, the GPU and CPU need to communicate to perform the update process. This approach increased the overhead in the initialization and update steps, which led to a low speedup. By comparison, in one of the experiments we conducted using our CUDA PSO implementation, we had more than 20,000 variables and were still able to achieve a speedup factor in excess of 300.
(2) In addition, the authors in [5] relied on pregenerated random numbers stored in texture memory for use when updating the particles based on velocity equations. However, storing random numbers in texture memory is not an efficient approach for large-scale problems. We describe a more efficient approach consisting of a device function to generate random numbers from inside the GPU code using the parallel Park–Miller algorithm.

To be able to achieve high performance, our mapping of PSO to GPU architectures carefully follows these steps:

- Map the tasks into multiple threads.
- Manage access to global memory to guarantee coalesced memory access.
- Manage access to shared memory.
- Manage thread synchronization.

We will now describe key aspects of our implementation with respect to these objectives.

4.1. Swarm structure in compute unified device architecture

Because there are several types of memory available on the GPU device, it is important to choose the appropriate memory type to use for each of the model's constituent elements. In our PSO model, positions and velocity variables are stored using a *struct* in CUDA. As depicted in Figure 12, a single array stores all variables of the same type for all particles. In the Memory Coalescing subsection, we described the reason for such an organization. Because shared memory is fast and can store modest amounts of data, we used it to store the fitness value, best local fitness value, and random number generator seed for each particle. In addition, we made sure our design made use of efficient instruction sequences that ensure concurrent tasks among all threads. For example, to reduce divergent branch operations as much as possible, we replaced this if–else statement:

```
if a<b then a=c else a=d
```

by using a single mathematical statement

```
a=(a<b)*c+(a>=b)*d
```


DOI: 10.1002/dac

```
typedef struct
{
        int X_SD[SD*popSize];
        float V_SD[SD*popSize];
        int p_SD[SD*popSize];
        int g_SD[SD];

        float X_Power [N*N*NUM_TRX*popSize];
        float V_Power[N*N*NUM_TRX*popSize];
        float p_Power[N*N*NUM_TRX*popSize];
        float g_Power[N*N*NUM_TRX];

        float X_Beam[N*N*NUM_TRX*popSize];
        float V_Beam[N*N*NUM_TRX*popSize];
        float p_Beam[N*N*NUM_TRX*popSize];
        float g_Beam[N*N*NUM_TRX];

        float  bestSwarmFitness;

}CUDA_SWARM;
```

Figure 12. Swarm structure in CUDA.

4.2. Particle swarm optimization coalesced memory access

The manner and frequency with which global memory is accessed has major performance implications for CUDA GPU implementations. Because of this, we carefully structured the PSO velocity and position variables in the global memory (as shown in Figure 13) to guarantee coalesced memory access. More specifically, instead of storing the variables associated with each particle one after the other, we grouped the value of each variable (for all particles) together. If the size of each variable is 4 bytes (‘*float*’ or ‘*integer*’), this structure ensures global memory coalescing, 16 threads (a half-warp) accessing 64 bytes of memory, simultaneously in one transaction.

4.3. Particle swarm optimization shared memory

In CUDA-enabled GPU devices, accessing to the shared memory by threads in the same warp is as fast as accessing a register, provided we guarantee that there is no bank conflict between the

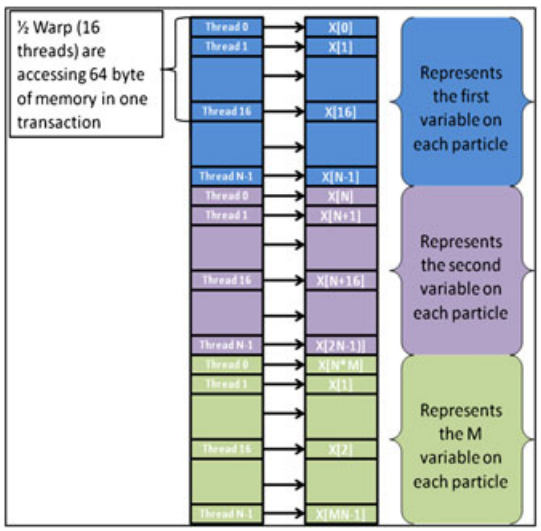

Figure 13. PSO variable representation to guarantee coalesced global memory access.

DOI: 10.1002/dac

threads [11]. In spite of its size limitations (16 KB) on each multiprocessor, we were able to use shared memory to store the values of: (i) particle fitness; (ii) best local particle fitness; and (ii) seeds used in the random number generator. With our approach, we guarantee that no shared memory bank conflict can arise because each thread is accessing its own variable that is located in a unique shared memory address. The schematic depicting the software/memory architecture is illustrated in Figure 14. Notice that every particle has one unique initial seed to generate its random numbers.

4.4. Random number generator

Because the GPU is not equipped with random number generators, several options are possible. One possible solution (e.g., considered in [5]) is to pregenerate all required random numbers using the CPU and store them in the GPU global memory or texture memory. The other option is to generate limited random numbers and store them in a shared memory and keep switching between CPU and GPU when there is a need for more random numbers. These two options are not performance-efficient because of the CPU–GPU communication overhead and the expensive memory access to the device memory. In [5], the authors used texture memory to store pregenerated random numbers. Using texture memory to store random numbers can hinder the performance of any CUDA implementation that requires many random numbers. This is because texture memory is effectively mapped to global memory and exhibits low bandwidth if there is a cache miss in the texture fetches.

In our PSO architecture, our approach to handle the random number generator is to implement a device function on the GPU to generate random numbers based on the Park–Miller random number generator [21]. In this implementation, we start with an initial seed for each thread stored in the shared memory. Figure 15 illustrates the implementation of this function.

4.5. Parallel single instruction multiple thread particle swarm optimization algorithm

In our proposed parallel SIMT PSO algorithm, we have multiple swarms that are executed simultaneously, each with its particles flying in the search space independently. Each swarm is represented by a block in the kernel grid and each particle in the swarm is executed by a separate thread. Our algorithm is detailed as follows:

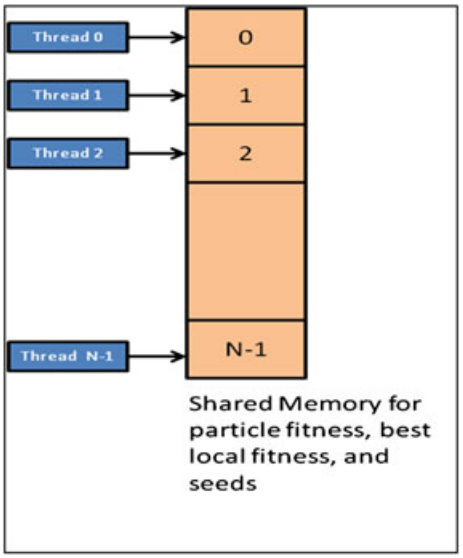

Figure 14. Shared memory representation to guarantee no bank conflict.

DOI: 10.1002/dac

Parallel SIMT PSO Algorithm

PSO kernel

```
Input: Initialzed array of Swarms d_S[NUM_SWARM]
Output: Updated Swarm using PSO
1.  Idx ← threadIdx.x
2.  __shared__ int seed[popSize]
3.  __shared__ float particlesFitness[popSize]
4.  __shared__ float bestParticleFitness[popSize]
5.  w←float value;
6.  c1←float value;
7.  c2←float value;
8.  If (idx<popSize) then
9.     particlesFitness[idx]←0
10.    seed[idx]← unique prime number
11.    bestParticleFitness[idx]= calcFitness(idx)
12. End if
13. For each iteration do
14.    If (idx<popSize) then
15.      updateParticle(idx,&d_S[blockIdx.x)
16.      particlesFitness[idx]=
    calcFitness(idx,&d_S[blockIdx.x)
17.      updateBestLocal(bestParticleFitness,
            particleFitness,idx,&d_S[blockIdx.x]);
18.    End if
19.    If (idx==0)then
20.      updateBestSwarm(particlesFitness
    ,&d_S[blockIdx.x ])
21.    End If
22. End for
```

Within the PSO kernel, each of the threads that execute a kernel has a unique thread ID represented by the *threadIdx* variable

updateParticle: Device function updates velocities and positions using SIMT

```
Input: current velocities and positions, and
thread idx
Output:Updated velocities and positions
1.  Index←idx
2.  For each power and beam variables do
3.       Update d_S->V_Power[index]
4.       Update d_S->X_Power[index]
5.       Update d_S->V_Beam[index]
6.       Update d_S->X_Beam[index]
7.       Index←Index+popSize
8.  End for
9.  Index←idx
10. For each SD velocities and positions do
11.      Update d_S->V_SD[index]
12.      Update d_S->X_SD[index]
13.      Index←Index+popSize
14. End for
```

updateBestLocal: Device function updates velocities and positions of best local fitness if the new fitness is minimum than current best local

updateBestSwarm: Device function updates velocities and positions of best swarm using the lowest fitness particle if it is less than current best swarm fitness

DOI: 10.1002/dac

```
__device__ double  d_intrnd(int * seed)
{
  double const a = 16807; //ie 7**5
  double const m = 2147483647; //ie 2**31-1

  double const reciprocal_m = 1.0/m;
  double temp = (*seed) * a;

*seed = (int)(temp - m * floor(temp *reciprocal_m));

return (double) ((double)*seed/(double)4294967295);

}
```

Figure 15. The Park–Miller random number generator.

5. PERFORMANCE RESULTS

In this section, we provide some experimental results to illustrate the performance of the proposed PSO technique for the hybrid RF/FSO topology control problem in large-scale networks. The test environment used in all experiments in this section is an Intel Quad Core 2.67 GHz CPU running Windows 2003 server with 6 GB memory. The GPU device is an NVIDIA GeForce GTX 285 with 240 processors (1.4 GHz) and 1 GB memory. The CUDA toolkit version is 2.1 with driver 181.20.

We started by running the experiment presented in our work in [17] to compare the optimal result obtained from the ILP versus the results obtained by our proposed PSO for the requested connections given in Table I. We ran the experiment under the same environment conditions, where we assume that the capacity of the FSO channel is 500 Mb/s, the capacity of the RF channel is 50 Mb/s, the FSO receiver sensitivity is –43 dBm, the RF receiver sensitivity is –84 dBm, and the maximum beam opening is 240 mrad.

The results presented in Tables II and III show that PSO provides a solution that is close to optimal where the total consumed power was 45 mW in the PSO solution, while the optimal solution was 35 mW. Furthermore, the selected beam-opening and channel assignment have close accuracy compared with the optimal selection to meet the joint throughput and end-to-end delay requirements.

In the following experiments, we placed 30 nodes in a 150 m× 150 m square area. The generated requests generated between all possible source–destination pairs are with equal probabilities resulting in having all source and destination nodes of all requests to be chosen with uniform probabilities. Our simulation tool generates n requests to determine the blocking probability of the network and the total consumed power by each transceiver in the network. The performance of our proposed PSO algorithm is evaluated for 30 swarms with a population size of 128 particles in each swarm, and the total number of iterations is 1000.

We compared our proposed PSO algorithm with the first-fit heuristic because of the simplicity of this heuristic. Furthermore, it was demonstrated in the literature that the first-fit heuristic produces

Table I. Traffic used to generate the topology.

	Source	Destination	Throughput (Mb/s)	Delay
1	1	2	5	1
2	1	5	5	1
3	2	4	100	2
4	2	5	100	1
5	3	1	250	1
6	4	3	5	1
7	4	2	5	2
8	5	4	100	1

DOI: 10.1002/dac

Table II. Routing and channel selection for each requested connection in PSO versus ILP.

Scenario	S	D	Route	Selected channels
ILP	1	2	1 → 2	0
	1	5	1 → 5	0
	2	4	2 → 5 → 4	2 → 1
	2	5	2 → 5	2
	3	1	3 → 1	1
	4	3	4 → 3	0
	4	2	4 → 3 → 2	0 → 0
	5	4	5 → 4	1
PSO	1	2	1 → 2	0
	1	5	1 → 5	0
	2	4	2 → 4	1
	2	5	2 → 5	2
	3	1	3 → 1	1
	4	3	4 → 3	0
	4	2	4 → 2	1
	5	4	5 → 4	3

Table III. Transmitted power and beam opening solution in PSO versus ILP.

Scenario	Link, transceiver	Transmitted power (mWat)	Beam opening (mrad)	Total consumed power
ILP	(1 → 2,0)	5	—	35
	(1 → 5,0)	5	—	
	(2 → 5,2)	5	80,240	
	(3 → 1,1)	10	80,80	
	(3 → 2,3)	5	80,160	
	(4 → 3,3)	5	80,80	
	(5 → 4,1)	5	80,160	
PSO	(1 → 2,0)	5	—	45
	(1 → 5,0)	5	—	
	(2 → 4,1)	5	80,240	
	(2 → 5,2)	5	80,160	
	(3 → 1,1)	10	80,80	
	(4 → 2,1)	5	80,80	
	(4 → 3,0)	10	—	
	(5 → 4,3)	5	80,160	

low blocking probabilities [22]. The proposed heuristics were compared in terms of their blocking probability and total transmission power by all nodes in the network.

The first experiment focused on the impact of increasing the number of connections in the hybrid RF/FSO networks. Figure 16 shows that the blocking performance of our PSO algorithm is performing much better than that of the first-fit heuristic under the different traffic loads. Also, by

DOI: 10.1002/dac

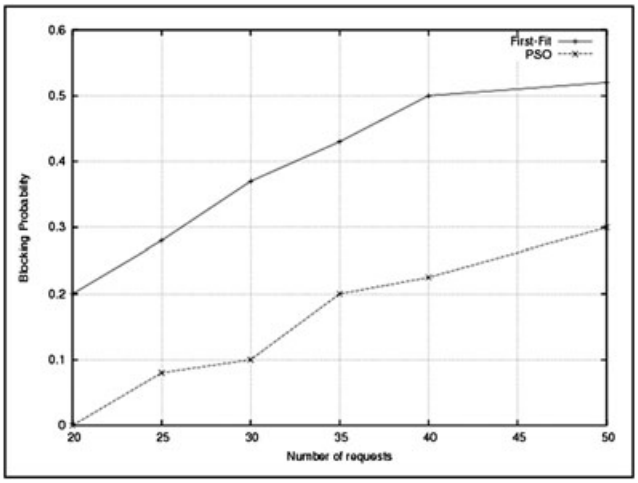

Figure 16. Blocking probability versus number of connections (N=30 and transceivers=5).

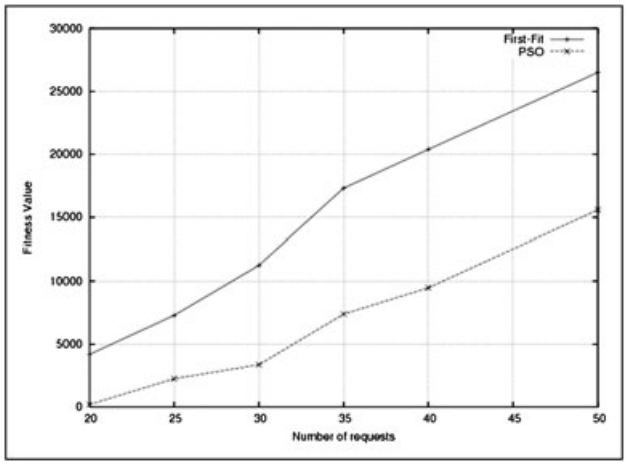

Figure 17. Fitness value versus number of connections (N=30 and transceivers=5).

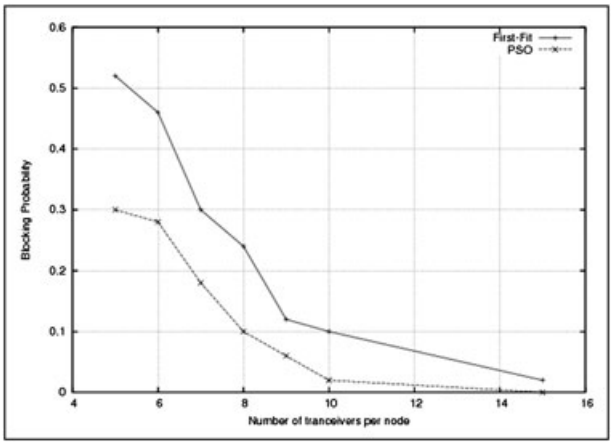

Figure 18. Blocking probability versus number of transceivers (N=30 and connections=50).

DOI: 10.1002/dac

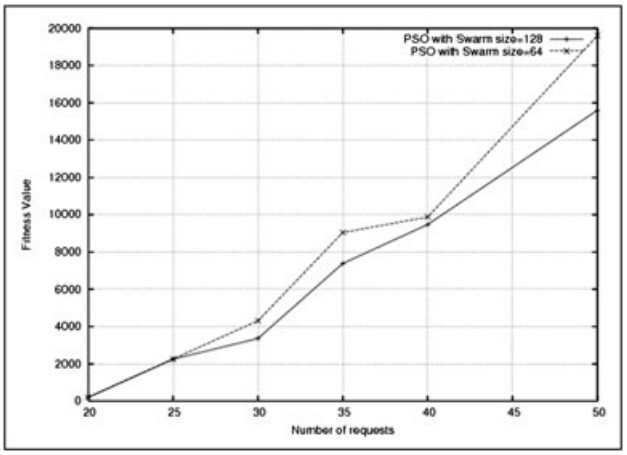

Figure 19. Fitness value versus number of transceivers using different swarm size (N=30 and connections=50).

Table IV. Parallel PSO using GPU speedup compared with sequential PSO implementation.

#Connections	Sequential PSO CPU time (s)	Parallel PSO CPU-GPU time (s)	Speedup
20	1221.28	3.11	392.51
25	1222.87	3.34	366.57
30	1244.58	3.46	359.78
35	1250.86	3.56	351.44
40	1269.41	3.65	348.15
50	1295.81	3.85	336.88

looking at the slopes, we conclude that the blocking probability increases as the connection set size increases. For example, 10% of the connections did not meet the QoS requirement using the PSO when the connection set size equaled to 30. Moreover, Figure 17 shows that the total fitness value, which includes the total transmission power used in our proposed PSO solution, is better than that of the first-fit heuristic.

The second experiment focuses on the impact of increasing the number of FSO transceivers on the blocking probability. The network is constructed in the same manner as before, but we varied the number of transceivers and we used a fixed number of connections, 50. By looking at the slopes in Figure 18, we conclude that the blocking probability percentage improves as we increase the number of FSO transceivers.

The third experiment focuses on the effect of the swarm size on the best fitness value. As demonstrated in Figure 19, the PSO with a swarm size equal to 128 particles was able to achieve a better solution compared with a swarm size of 64 particles under a different traffic load.

The last experiment evaluates the speedup achieved using the parallel PSO implementation in the CUDA-enabled GPU compared with the sequential implementation in CPU. As Table IV illustrates, the GPU implementation achieved substantial speedup ranging from 336× to 392× in all experiments we conducted.

6. CONCLUSION AND FUTURE WORK

We developed a parallel PSO model to solve large-scale topology control problems in hybrid RF/FSO mesh networks. The strength of the proposed PSO solution stems from its simplicity, applicability to large-scale networks, and its efficiency compared with other heuristics proposed in the

DOI: 10.1002/dac

literature. Another advantage of using the PSO approach is that it is a general technique requiring few parameter settings. We compared the total transmission power and blocking performance of the solutions obtained by PSO with those obtained using the first-fit heuristic, and we showed that the PSO solutions exhibit a significantly better performance. We improved our PSO implementation's efficiency by mapping it to a CUDA-enabled GPU, the NVIDIA GeForce GTX 285. The GPU-based implementation exhibited a speedup ranging from 336× to over 392× relative to the sequential CPU implementation. Innovations in the memory/execution structure of our mapping enabled us to surpass all prior known PSO GPU implementations. Our results provide a promising indication of the viability of GPU-based approaches to large-scale optimization problems, such as those found in RF/FSO wireless mesh network designs.

In the future, we will extend our model to include an adaptive Forward Error Correction (FEC) and Automatic Repeat Request (ARQ) protocol for hybrid RF/FSO networks. The operational parameters of these protocols will be part of a new, larger optimization problem (added to those considered in this work). We will seek to determine the optimal values of the parameters in these much larger optimization problems using PSO optimization on CUDA-enabled GPUs, and thereby assess the scalability of our current approaches to more complex optimizations.

REFERENCES

1. Kennedy J, Eberhart R. Particle swarm optimization. *IEEE International Conference on Neural Networks (Perth, Australia)*, IEEE Service Center, Piscataway, NY, IV: 1995; 1942–1948.
2. Available from: http://www.swarmintelligence.org.
3. Melanie M. *An Introduction to Genetic Algorithms*. MIT Press: Cambridge, MA, 1996.
4. Cai X, Cui Z, Zeng J, Tan Y. An efficient fine-grained parallel Particle Swarm Optimization method based on GPU-Acceleration. *International Journal of Innovative Computing, Information and Control (ICIC)* 2007; **3**(6(B)).
5. Zhu W, Curry J. Particle Swarm with Graphics Hardware Acceleration and Local Pattern Search on Bound Constrained Problems. *IEEE Swarm Intelligence Symposium*, Nashville, TN, USA, 2009.
6. Laguna-Sanchez G, Olguín-Carbajal M, Cruz-Corts N, Barrn-Fernndez R, lvarez-Cedillo J. Comparative Study of Parallel Variants for a Particle Swarm Optimization Algorithm Implemented on a Multithreading GPU. *Journal of Applied Research and Technology (JART)* 2009; **7**(3):292–309.
7. Preis T, Virnau P, Paul W, Schneider J. GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model. *Journal of Computational Physics* 2009; **228**:4468–4477.
8. Januszewski M, Kostur M. Accelerating numerical solution of Stochastic Differential Equations with CUDA. *Computational Physics* 2009. Available from: http://www.citebase.org/abstract?id=oai:arXiv.org:0903.3852
9. Available at http://www.lapix.ufsc.br/fibertracking/
10. Stone S, Haldar J, Tsao S, Hwu W, Liang Z, Sutton B. Accelerating Advanced MRI Reconstructions on GPUs. In *Proceedings of 5th International Conference on Computing Frontiers*. ACM: New York, NY, May 2008.
11. NVIDIA. NVIDIA CUDA programming manual, December 2008.
12. Available from: http://en.wikipedia.org/wiki/Particle_swarm_optimization.
13. Clerc M. *Particle Swarm Optimization*. ISTE: London, UK, 2006.
14. Willebrand H, Ghuman BS. *Free-space Optics: Enabling Optical Connectivity in Today's Networks*. SAMS Publishing: Indianapolis, IN, 2002.
15. Ben Brahim G, Awwad O, Al-Fuquaha A, Khan B, Kountanis D, Guizani M. A New MILP Formulation of a New Budgeted Location-Based Cooperative Model for MANETs. *IEEE Globecom*, Washington DC, Nov. 26–30, 2007.
16. Jaime L, Aniket D, Eswaran B, Stuart M, Christopher D. Optimizing performance of hybrid FSO/RF networks in realistic dynamic scenarios. *Free-Space Laser Communications V, Proceedings of the SPIE,Vol 5892*, 2005; 52–60.
17. Awwad O, Al-Fuqaha A, Kountanis D, Rayes A. Topology Control using Adaptive Power Control and Beam-Width in Hybrid RF/FSO MANETs, CHINACOM 2008, August, 2008.
18. Awwad O, Al-Fuqaha A, Rayes A. Traffic grooming, routing, and wavelength assignment in WDM transport networks with sparse grooming resources. *Computer Communications* 2007; **30**:3508–3524.
19. Parsopoulos K, Vrahatis M. Particle swarm optimization method for constrained optimization problems. *Frontiers in Artificial Intelligence and Applications* 2002; **76**:215–220.
20. Eberhart R, Simpson PK, Dobbins W. *Computational Intelligence PC Tools*. Academic Press Professional: Boston, MA, 1996.
21. Langdon W. A fast high quality pseudo random number generator for graphics processing units. In *IEEE World Congress on Computational Intelligence (Hong Kong)*, J Wang (ed.). IEEE; 459–465.
22. Simmons J, Goldstein E, Saleh A. Quantifying the Benefit of Wavelength Add-Drop in WDM Rings with Distance Independent and Dependent Traffic. *IEEE/OSA Journal of Lightwave Technology* 1999; **17**:48–57.

DOI: 10.1002/dac

AUTHORS' BIOGRAPHIES

Osama Awwad received his M.S. and Ph.D. degrees in Computer Science from Western Michigan University, in 2006 and 2009, respectively, and a B.S. degree in Computer Engineering from Jordan University of Science and Technology in 2003. Currently, he is a Performance Monitoring Consultant at Quest Software. His research interests include enhancing the communication infrastructure in wireless (radio frequency + free space optics) networks, high performance computing using CUDA/GPU-based computing, game theory, QoS routing, WDM optical networks design, and application performance management.

Ala Al-Fuqaha (S'00-M'04) received his M.S. and Ph.D. degrees in Electrical and Computer Engineering from the University of Missouri, in 1999 and 2004, respectively. Currently, he is an Associate Professor at the Computer Science Department of Western Michigan University. Before joining Western Michigan University, Dr. Al-Fuqaha was a senior member of the technical staff at Lambda Optical Systems in Reston, VA where he worked on the design and development of embedded routing protocols and network management systems. Prior to Lambda, Dr. Al-Fuqaha was a software engineer with Sprint Telecommunications Corp. where he worked as part of the architecture team. His research interests include intelligent network management and planning, QoS routing and performance analysis, and evaluation of high-speed computer and telecommunication networks. Dr. Al-Fuqaha has served as a Technical Program Committee member of many IEEE conferences.

Ghassen Ben Brahim (S'04) received his B.Sc. degree in Computer Science from the National School of Computer Science in Tunisia, M.S. degree from the University of Missouri-Columbia, and Ph.D. degree from Western Michigan University. His research interests include wireless networks, QoS routing in large-scale mobile ad hoc networking, routing in all-optical networks, network performance analysis, and design and analysis of network protocols. He joined the Naval Research Laboratory in Washington DC in January 2000, where he worked as a software engineer in the ATDNET/MONET project. In January 2001, he joined Lambda Optical Systems as a member of the technical staff. He was involved in the design and implementation of line and path protection frameworks for all-optical switches based on the microelectromechanical systems technology and the design and implementation of the extensions for the Open Shortest Path First (OSPF) routing protocol to provide QoS and source routing in all-optical networks.

Bilal Khan is Professor of Mathematics and Computer Science at John Jay College, City University of New York where he is a doctoral faculty in the programs of Computer Science, Forensic Computing, and Criminal Justice. He received his B.Sc. from MIT (1993), M.S. from Johns Hopkins (1997) and Ph.D. from City University of New York (2003). He is the author of over 75 refereed journal and conference publications on the theoretical and empirical aspects of networks.

DOI: 10.1002/dac

Ammar Rayes received his Doctor of Science degree in Electrical Engineering from Washington University in St. Louis, MO, in 1994. Dr. Ammar Rayes is a Senior Manager at the Advanced Technology Support group at Cisco Systems. He has managed several solutions including Customer Advocacy Platform Architecture, wireless, broadband access, subscriber and security management, triple play over Ethernet, virtual private network, performance and traffic engineering, IOS embedded management, and simple network management protocol infrastructure and tools. Prior to joining Cisco Systems, he was a director at the Traffic Capacity Management and Planning Department at Telcordia Technologies. Dr. Rayes has authored/co-authored over 50 patents and papers on advances in numerous communications-related technologies including a book on asynchronous transfer mode switching and network design (McGraw Hill-1999). He is currently chairing the Cisco CA Patent Council and serving on the board of the International Journal of Software Architecture and Network Management Research Council.

DOI: 10.1002/dac