Implementing Central Force Optimization on the Intel Xeon Phi

Thomas Charest & Robert C. Green II
Dept. of Computer Science
Bowling Green State University
Bowling Green, OH 43402
Email: {charest,greenr}@bgsu.edu

***Abstract*—Central Force Optimization (CFO) is a fully deterministic population based metaheuristic algorithm based on the analogy of classical kinematics. CFO yields more accurate and consistent results compared to other population based metaheuristics like Particle Swarm Optimization and Genetic Algorithms, but does so at the cost of higher computational complexity, leading to increased computational time. This study presents a parallel implementation of CFO written in C++ using OpenMP as implemented for both a multi-core CPU and the Intel Xeon Phi Co-processor. Results show that parallelizing CFO provides promising speedup values from 5-35 on the multi-core CPU and 1-12 on the Intel Xeon Phi.**

***Index Terms*—Central Force Optimization, Metaheuristic, Parallel, Xeon Phi, Multi-core**

I. Introduction

Central Force Optimization (CFO) is a 'deterministic multi-dimensional search metaheuristic based on the metaphor of gravitational kinematics' [1]. A metaheuristic is a generalized algorithm that finds the optimal values for a given optimization problem, typically in the form of a function. Other popular metaheuristics include Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO), and Genetic Algorithms (GA). In contrast to CFO, other metaheuristics are typically stochastic – having some inherent form of pseudo-random behavior – while CFO is completely deterministic. The deterministic nature of CFO has resulted in many advantages including the necessity of fewer function evaluations [2] as the algorithm has no need of multiple runs.

Along with the advantages of determinism, the fundamental metaphor that CFO is based on (gravitational kinematics), comes with the challenge of increased computational complexity, resulting in extended computation time. CFO's computation time has been addressed in the past through the implementation of the algorithm on a Graphics Processing Unit (GPU) [3]–[7]. While substantial, these studies lack the evaluation of any implementation on a common, multi-core, commodity processor or any other modern, parallel computing platform. As such, this paper presents a case study regarding the parallelization of the CFO algorithm using OpenMP on two platforms – a typical multi-core processor and the Intel Xeon Phi. The remainder of this paper is structured as follows: Section II reviews the fundamentals of the CFO algorithm; Section III discusses the implementation of this study; Section IV show the results of this study; and Section V concludes this study.

978-1-7281-7445-7/20/$31.00 © 2020 IEEE
DOI 10.1109/IPDPSW50202.2020.00091

II. Central Force Optimization

The fundamental, natural metaphor that is used in CFO is that of probes flying through space. As these probes travel, they are moved by gravitational forces of each other as well as other encountered objects. The typical equations used to model such behavior have been modified to accommodate this metaphor computationally according to (1)–(3) where

F is force between two masses, M is mass, R is position, A is acceleration, p is the current probe, k is another probe, j is the current time step, N_p is the total number of probes, G is the gravitational constant, α and β are constants, and U is unit step function.

$$F = M_{j-1}^{k} - M_{j-1}^{p} \tag{1}$$

$$A_{j-1}^{p} = G \sum_{k=1, k\neq p}^{N_p} U(F) \cdot F^{\alpha} \frac{(R_{j-1}^{k} - R_{j-1}^{p})}{|(R_{j-1}^{k} - R_{j-1}^{p})|} \tag{2}$$

$$R_{j}^{p} = R_{j-1}^{p} + \frac{1}{2} A_{j-1}^{p} \Delta t^{2} \tag{3}$$

Based on these equations, the fundamental CFO algorithm can be described as follows [1], [3]:

- Initialize position of all probes
- Initialize acceleration of all probes
- Calculate initial fitness
- Record best fitness value
- Until stopping condition is met:
  - Update each probe's position
  - Retrieve errant probes
  - Calculate fitness values of each probe
  - Update best fitness value
  - Compute new acceleration for each probe

Of note is the step "Retrieve Errant Probes". In this step, the CFO algorithm prunes those probes whose position has gone outside of the search space according to (4) and (5) where i is the current dimension, X_{min} and X_{max} are vectors containing minimum/maximum values in each dimension, and F_{rep} is the reposition factor, usually set to 0.5 [3].

$$R(p,i,j) = X_{min} + F_{rep} * (R(p,i,j-1) - X_{min}(i)) \quad (4)$$

$$R(p,i,j) = X_{max} + F_{rep} * (X_{max}(i) - R(p,i,j-1)) \quad (5)$$

This fundamental variant of CFO has also undergone a variety of modifications (including Pseudo Random CFO (PR-CFO) [8]–[10], Parameter Free CFO (PF-CFO) [11], Distributed, Multi-Objective CFO [12], and Adaptive CFO [13]), been applied to a variety of problems (Antenna Benchmarking [1], [14], [15], training Neural Networks [2], [12], [13], iris recognition [13], and optimizing drinking water networks [16]–[18]), and undergone analyses ([3].

In addition, the algorithm has previously been implemented on Graphics Processing Units (GPUs) [3], [5], [6].These studies have all proposed implementations of CFO on the GPU, and some compare the performance of CFO to other metaheuristics [6]. The benchmarking studies in [3] exhibit speedup values in the range of 1 to 28, depending on certain conditions, and further investigates the runtime bottlenecks within the CFO algorithm. It is also noted that “updateAcceleration” requires the highest average computation time (99.94% of all time spent), as well as the highest complexity: $N_p^2 \times N_d^2$.

A. OpenMP

OpenMP (Open Multi Processing) is an open-source, cross-platform API used to parallelize shared memory applications. OpenMP is utilized in a C/C++ application by the use of compiler directives. In parallel sections, OpenMP employs a thread-based “fork-join” model [19], which spawns a linked hierarchy of threads from a singular main thread. Once out of a parallel section, these threads join back to a singular thread. There are a number of performance considerations for shared memory parallel applications, namely: shared data access, shared memory synchronization, and sequential consistency.

B. Intel Xeon Phi

The Intel Xeon Phi is a PCI-e expansion card on the host system that contains an autonomous many-core CPU. To utilize this card, the host system can either transfer an executable compiled for the architecture and run natively on the Xeon Phi, or it could send work from the host CPU to the coprocessor in an offloading model. The particular Xeon Phi card studied by the author has 57 cores, with a suggested four hardware threads per core for saturation, totaling an ideal 228 hardware threads.

III. Methodology

This study parallelizes an implementation of the CFO algorithm written in C++ using OpenMP [1]. These benchmarks were first executed on a host system, and then compiled for native execution on the Intel Xeon Phi coprocessor. The host system features two Intel Xeon E5-2620 6-core processors, running CentOS 6.7, and 32GB RAM.

[1]https://gitlab.com/central-force-optimization/oo-central-force-optimization/tree/OpenMP

In the scope of CFO, whose processing is neatly segmented into separate steps, it was imperative to identify portions of the CFO algorithm that were the most computationally expensive. Prior work on parallel implementations of CFO (in this case, using CUDA) shows that the `updateAcceleration` function takes 99.94% of CFO’s total computation time [3], making it a prime candidate for parallelization.

For an effective parallel implementation that increases performance, one must keep in mind a number of things: shared data access, shared memory synchronization, and sequential consistency. From the code snippet in Listing 1, one can see each of these issues come to light. Although each probe’s acceleration can be calculated independently in a parallel environment, the shared memory access must be pointed out on the `calculateInfluence` method call (which calculates the influence between each probe i and j), and the initialization of a probe’s acceleration on line 5. This is imperative for an accurate parallel implementation, and a necessity that may have an effect on performance. Other performance considerations in the “calculateInfluence” method arise, particularly shared memory synchronization and consistency as the acceleration is updated with each call probe by probe, and acceleration must be available to other parallel threads.

In addition,

The `updateAcceleration` function also reveals a granularity problem for parallel computation- i.e. which “for” loop should be parallelized? The author executed preliminary benchmarks on separate OpenMP implementations of outer (line 3), middle (line 4), and inner (line 6) “for” loop parallelization. The most appropriate parallel implementation was revealed to be the outer loop implementation, boasting the least wasted computation time spent on waiting and communication. The modified version of the updateAccel function used in this study is shown in Listing 2.

To preserve benchmarking data for later analysis, the author created a database logging system as an extension of previous logging systems of the CFO algorithm. The database logging system stores input parameters to the algorithm, interpreted and calculated parameters (from PF-CFO, if applicable) and finally the runtime of various sections of execution. The database contains nearly complete runtime information for the algorithm including various timings and counts. All figures and metrics are calculated from the total runtime of the algorithm.

Listing 1: updateAcceleration function

```
1  void updateAcceleration(int j){
2    int p, i, k;
3    for ( p = 0; p < Np; p++) {
4      for ( i = 0; i < Nd; i++) {
5        A[p][i][j] = 0;
6        for ( k = 0; k < Np; k++) {
7          if (k != p) {
8            calculateInfluence(p, i, k, j);
9          }
10       }
11     }
```

```
    }
}
```

Listing 2: updateAccel as Modified for the Xeon Phi

```
void updateAcceleration(int j){
  int p, i, k;

  #pragma omp parallel default(none)
  private(sumSQ, denom, numerator, alpha,
  beta, p, i, j, k)
  {
    #pragma omp for schedule(dynamic)
    for ( p = 0; p < Np; p++) {
      for ( i = 0; i < Nd; i++) {
        A[p][i][j] = 0;
        for ( k = 0; k < Np; k++) {
          if (k != p) {
            calculateInfluence(p,i,k,j);
          }
        }
      }
    }
  }
}
```

Along with the database logging capabilities, the author also modified the CFO implementation to accept command line input for various CFO parameters, and an option to run PFCFO prior to executing the CFO algorithm, and set certain parameters from the previous PFCFO algorithm. Among other options is the ability to set available threads equal to N_p, which is used on the Xeon Phi since it is the architecture capable of supporting up to 226 threads.

Benchmarking the performance of a metaheuristic requires testing functions, and thus this research tested 21 previously studied test functions for the CFO algorithm. These test functions are described in I. The CFO algorithm was also compiled for native execution on the Intel Xeon Phi coprocessor and evaluated for dimensions $Nd = \{10, 20, 30\}$, and for threads $10 - 240$ in intervals of 10. A final run of the CFO algorithm is executed on the Xeon Phi where the number of threads is equal to the optimal N_p for each test function.

IV. Results

A. OpenMP Results

Results from this study are grouped into two distinct groups, labeled groups A and B. This grouping is to facilitate the discussion of each distinct test function against other functions in the testing set. Test functions are grouped by relative performance increases due to thread count and dimensions. The intuition behind this grouping can be seen in table II, which shows the optimal N_p and γ (given a N_d) for every test function. Table II is the result of running PF-CFO algorithm on all test functions, logging optimal N_p and γ values, and benchmarking parallel CFO. Further, this table shows the group of each test function. Those with higher optimal N_p are in group B, while lower N_p values overall are placed in group A. Function f_{23} is placed in group B as its performance is closer to those in said group.

Figs. 1, 3 and 5 show the speedup and efficiency of the test functions in group A given the dimensions $Nd = 10, 20, 30$. Fig.1a shows the group A's sublinear speedup at $N_d = 10$, hitting a high of around 10 threads, with a speedup plateau for all functions beginning at 12 threads, leading to a maximum speedup of around 4.5 for this figure. The corresponding efficiency for group A at $N_d = 10$, Fig. 1b shows a steady drop of efficiency dropping to around 0.2 at 12 threads. In contrast, Fig. 2 shows a much higher speedup of around 7.5 for the majority of test functions, boasting somewhat linear speedup plateauing around 12 threads. Group B also shows better efficiency for all test functions, remaining greater than 0.5 for threads 1-12.

For $N_d = 20$, figure 3 shows that group A exhibits near linear speedup until a large spike in speedup after six threads for three separate functions. By 12 threads, most functions exhibit close to linear speedup. Fig. 4 shows a similar close to linear speedup, with efficiency greater than 0.75 until around 12 threads. Group B also exhibits a large spike at six threads for the majority of functions in the group, and a larger spike for one function resulting in the highest speedup of around 18 at 10 threads.

At $N_d = 30$, figure 5 shows speedup and efficiency of group A, and Fig. 5a exhibits the highest speedup in the study of around 35 at 12 threads. There is still a speedup spike for most functions at six threads, and further trends upwards in speedup for most functions. The corresponding graph of group A's efficiency shows superlinear speedup for all but three functions by 16 threads. Fig. 6 shows similarly impressive results for Group B, resulting in superlinear speedup for the majority of the functions as well.

Overall, both groups experienced either close to linear speedup, or superlinear speedup for some specific functions for higher values of N_d. The majority of test functions across both groups maintained efficiency values of 0.5 or above given Nd of 20 and above. Further, the speedup and efficiency of both groups increased at higher values of Nd. Group B performed consistently better at $N_d = 20$, while group A achieved better results for a select set of functions.

B. Xeon Phi Results

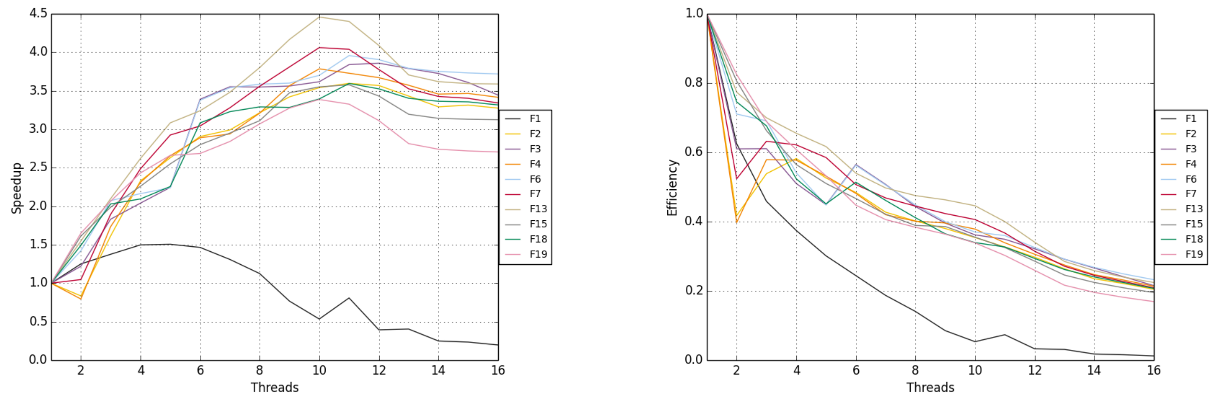
For consistency, the following benchmarks of the Intel Xeon Phi coprocessor have been similarly split into groups A and B. Both groups contain the same test functions as previously noted.

Although Speedup and Efficiency values are calculated against a serial run for a program, the Xeon Phi is calculated against a serial run on the host system, which may influence the efficiency measure as the Xeon Phi threads are not as powerful as those on the host system.

Test function performance on the Xeon Phi for $N_d = 10$ is relatively poor. Group A achieves minimal speedup, at most 2,

Function index	Function Name	Formula
f_1	Sphere	$f(x) = \sum_{i=1}^{N_d} x_i^2$
f_2	Rastrigin	$f(x) = 10N_d + \sum_{i=1}^{N_d}[x_i^2 = 10\cos(2\pi x_i)]$
f_3	Griewank	$f(x) = \sum_{i=1}^{N_d} \frac{x_i^2}{4000} - \prod_{i=1}^{N_d} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$
f_4	Rosenbrock	$f(x) = \sum_{i=1}^{N_d-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$
f_5	Ackley	$f(x) = -a\exp\left(-b\sqrt{\frac{1}{N_d}\sum_{i=1}^{N_d} x_i^2}\right) - \exp\left(\frac{1}{N_d}\sum_{i=1}^{N_d}\cos(cx_i)\right) + a + \exp(1)$
f_6	Schwefel 222	$f(x) = -\left[\sum_{i=1}^{N_d} \lvert x_i \rvert + \prod_{i=1}^{N_d}\right]$
f_7	Lunacek	$f = -\left(min\left(\sum_{i=1}^{N_d}(x_i - u_1)^2, d * N_d + s * \left(\sum_{i=1}^{N_d}(x_i - u_2)\right)\right) + 10 * \sum_{i=1}^{N_d} 1 - \cos(2\pi(x_i - u_1))\right),$ where $d = 1$, $s = 1 - \frac{1}{2\sqrt{N_d+20} - 8.2}$, $u_1 = 2.5$, $u_2 = -1\sqrt{\frac{u_1^2 - d}{s}}$
f_8	Ridge	$f(x) = \sum_{i=1}^{N_d}(\sum_{k=1}^{i} x_k)^2$
f_9	Schaffer's F7	$f(x) = -\sum_{i=1}^{N_d-1}\left(\frac{1}{N_d - 1}\sqrt{x_i^2 + x_{i+1}^2}\sin(50 * \sqrt{x_i^2 + x_{i+1}^2}^{0.2})\right)^2$
f_{10}	Beale	$f(x) = (1.5 - x_1 + x_1x_2)^2 + (2.25 - x_1 + x_1x_2^2)^2 + (2.625 - x_1 + x_1x_2^3)^2$
f_{11}	Booth	$f(x) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2$
f_{12}	Bulkin	$f(x) = 100\sqrt{\lvert x_2 - 0.01x_1^2 \rvert} + 0.01\lvert x_1 + 10 \rvert$
f_{13}	6 Hump Camel	$f(x) = \left(4 - 2.1x_1^2 + \frac{x_1^4}{3}\right)x_1^2 + x_1x_2 + (-4 + 4x_2^2)x_2^2$
f_{14}	Easom	$f(x) = -\cos(x_1)\cos(x_2)\exp\left(-(x_1 - \pi)^2 - (x_2 - \pi)^2\right)$
f_{15}	Goldstein-Price	$f(x) = [1 + (x_1 + x_2 + 1)^2(19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2)] \times [30 + (2x_1 - 3x_2)^2(18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)]$
f_{16}	Levi	$f(x) = \sin^2(\pi w_i) + \sum_{i=1}^{N_d-1}(w_i - 1)^2[1 + 10\sin^2(\pi w_i + 1)] + (w_d - 1)^2[1 + \sin^2(2\pi w_d)]$, where $w_i = 1 + \frac{x_i - 1}{4}$ for all $i = 1, \ldots, d$
f_{17}	Matya	$f(x) = 0.26(x_1^2 + x_2^2) - 0.48x_1x_2$
f_{18}	Modified Double Sum	$f = -\sum_{i=1}^{N_d}\sum_{k=1}^{i+1}(x_k - (k+1))^2$
f_{19}	Whitley	$f(x) = \sum_{i=1}^{N_d}\sum_{k=1}^{N_d}\frac{[100 * x_i^2 - x_k^2]^2}{4000}$
f_{20}	Rana	$f(x) = -\left(\left[x_1\sin(\sqrt{\lvert x_2 + 1 - x_1 \rvert}]\cos\sqrt{\lvert x_2 + 1 - x_1 \rvert}\right] + \left[(x_2 + 1)\cos(\sqrt{\lvert x_2 + 1 - x_1 \rvert})\sin(\sqrt{\lvert x_2 + 1 - x_1 \rvert})\right]\right)$
f_{21}	Schwefel	$f(x) = 418.9829N_d - \sum_{i=1}^{N_d} x_i\sin(\sqrt{\lvert x_i \rvert})$

TABLE I: Test Functions



(a) Group A Speedup

(b) Group A Efficiency

Fig. 1: Group A test functions for $N_d = 10$

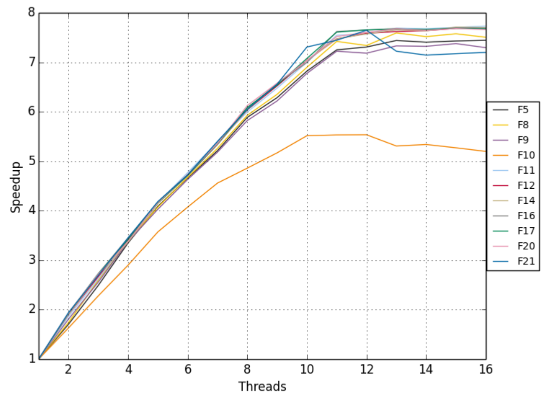
while group B achieves a more consistent speedup of around 2.5 for many more test functions than in group A.

At $N_d = 20$, figure 9 shows that group A continues its trend of scattered speedup, with the majority exhibiting speedup of $1-2$, and one function exhibiting a speedup of 8. Fig.10 shows group B exhibits consistently higher speedup values for most functions, topping off at 6 to 9.

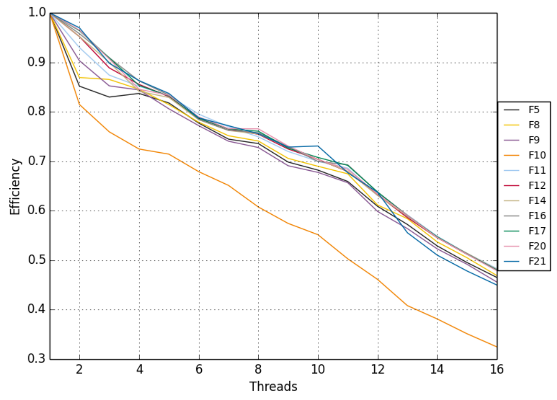
At $N_d = 30$, figure 11 exhibits relatively flat speedup after

Function index	N_p given $N_d = 10$	N_p given $N_d = 20$	N_p given $N_d = 30$	γ	Group
f_1	20	40	60	0.5	A
f_2	40	80	120	0.5	A
f_3	20	40	60	0.5	A
f_4	40	80	120	0.75	A
f_5	120	160	180	1.0	B
f_6	20	40	60	0.5	A
f_7	40	80	120	0.8	A
f_8	120	160	180	0.5	B
f_9	120	160	180	1.0	B
f_{10}	60	120	180	0.6	B
f_{11}	120	160	180	0.65	B
f_{12}	120	160	180	1.0	B
f_{13}	40	80	120	0.65	A
f_{14}	120	160	180	0.5	B
f_{15}	40	80	120	0.25	A
f_{16}	120	160	180	0.55	B
f_{17}	120	160	180	0.5	B
f_{18}	20	160	180	0.75	A
f_{19}	40	80	120	0.55	A
f_{20}	120	160	180	0.6	B
f_{21}	100	160	180	0.9	B

TABLE II: Optimal N_p and γ values



(a) Group B Speedup



(b) Group B Efficiency

Fig. 2: Group B test functions for $N_d = 10$

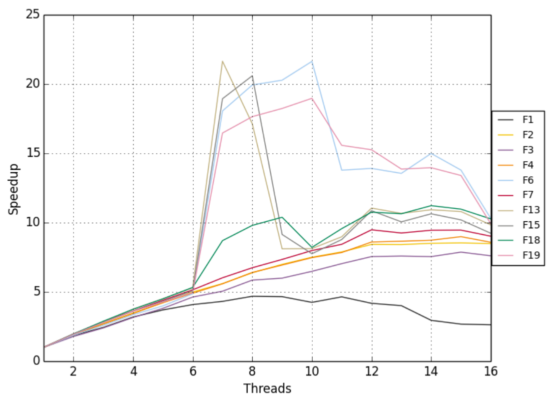
50 threads, with one outlier increasing to a speedup of 11 at 200 threads. In contrast to group A, figure 12 shows group B climbing in speedup in a step-like manner for all threads.

As for the Xeon Phi's relatively low speedup compared to the host system's performance, the discrepancy may come from the Xeon Phi architecture's lack of memory. Parallel CFO's shared memory synchronization and shared memory access is discussed earlier, and may have had an affect on the Xeon Phi's performance

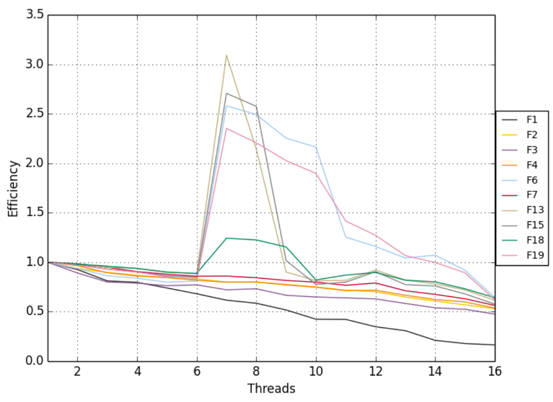
Further, the Xeon Phi implementation is simply a native execution on the coprocessor, and the algorithm may be better suited for an offloading implementation where the coprocessors are only invoked for particularly parallel portions of execution.

From previous figures, one can see that the speedup for each test function begins flattening at different points, particularly on the Xeon Phi. The plateauing of performance along with the updateAcceleration (Fig. 1) function's outer loop parallelization implies a direct relationship between N_p and speedup. Further, given that each test function was benchmarked with increasing values of N_d and optimal N_p (i.e. increasing with each N_d), it is fruitful to investigate the influence of N_p and speedup.

Fig. 13 is a scatter plot of calculated optimal N_p values for each function vs maximum speedup after benchmarking these functions on the Xeon phi. There is direct relationship

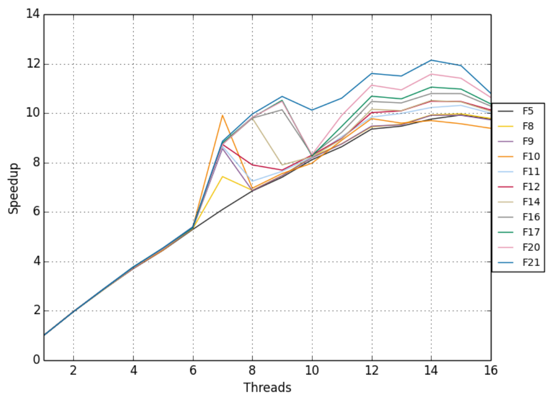


(a) Group A Speedup

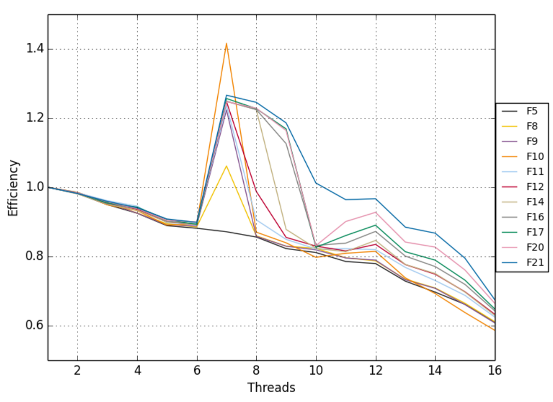


(b) Group A Efficiency

Fig. 3: Group A test functions for $N_d = 20$

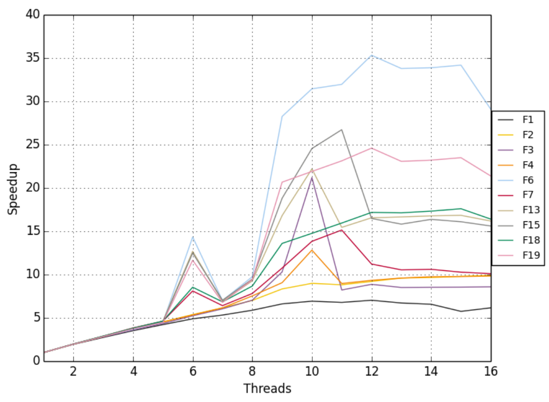


(a) Group B Speedup

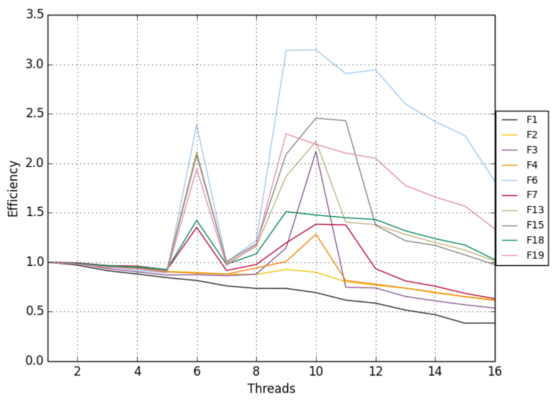


(b) Group B Efficiency

Fig. 4: Group B test functions for $N_d = 20$

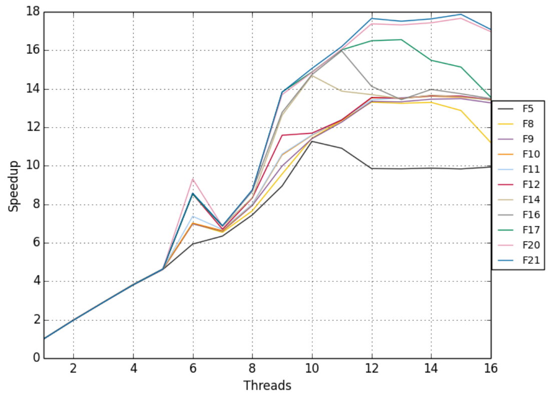


(a) Group A Speedup

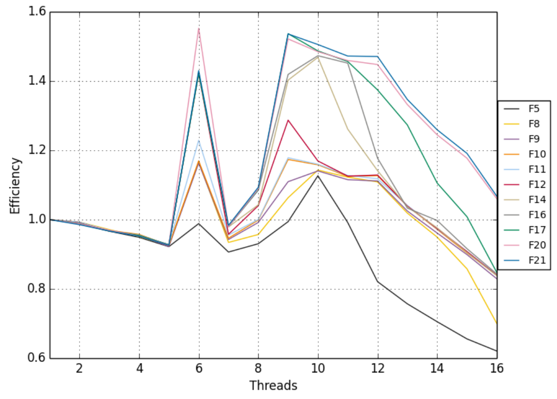


(b) Group A Efficiency

Fig. 5: Group A test functions for $N_d = 30$

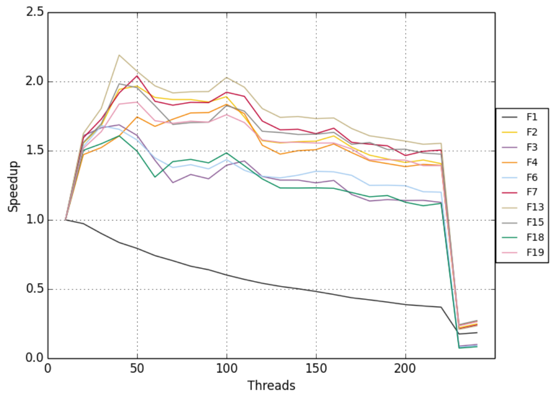


(a) Group B Speedup

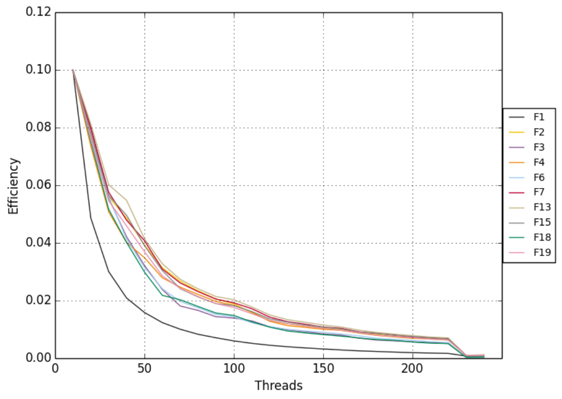


(b) Group B Efficiency

Fig. 6: Group B test functions for $N_d = 30$

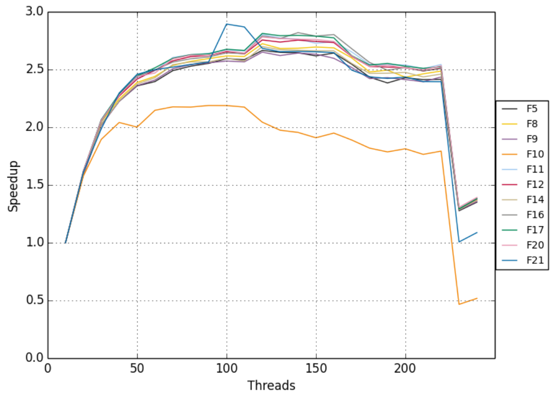


(a) Group A Speedup for Xeon Phi

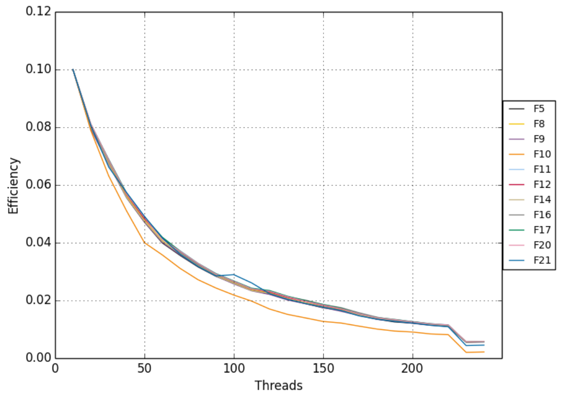


(b) Group A Efficiency for Xeon Phi

Fig. 7: Group A test functions for $N_d = 10$

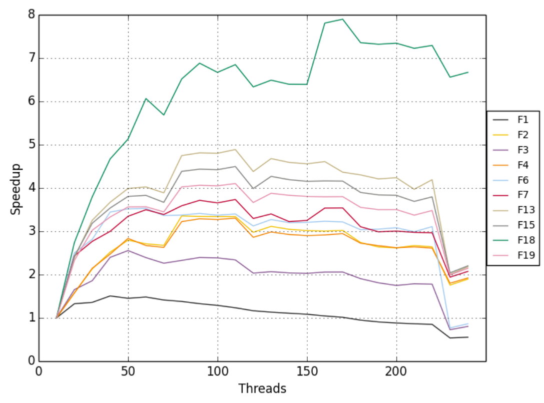


(a) Group B Speedup for Xeon Phi

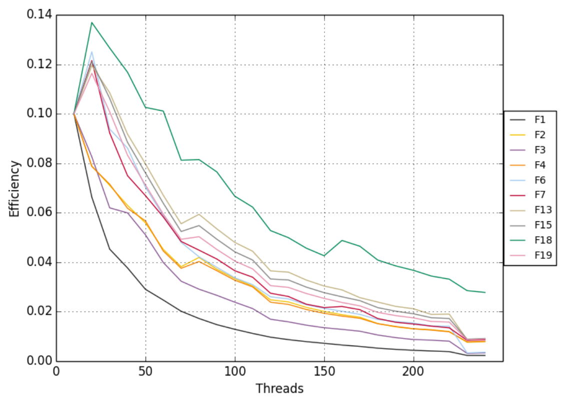


(b) Group B Efficiency for Xeon Phi

Fig. 8: Group B test functions for $N_d = 10$

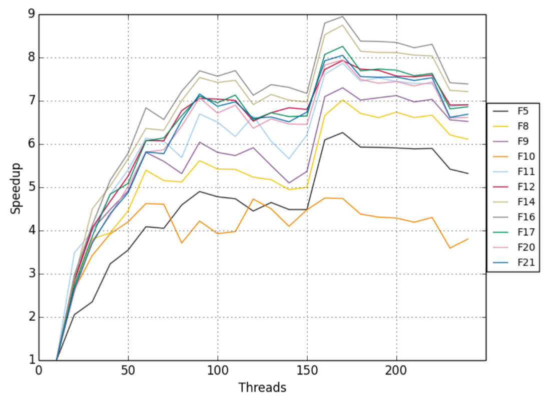


(a) Group A Speedup for Xeon Phi

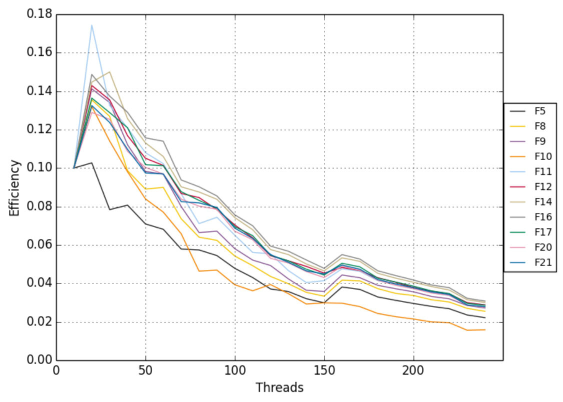


(b) Group A Efficiency for Xeon Phi

Fig. 9: Group A test functions for $N_d = 20$

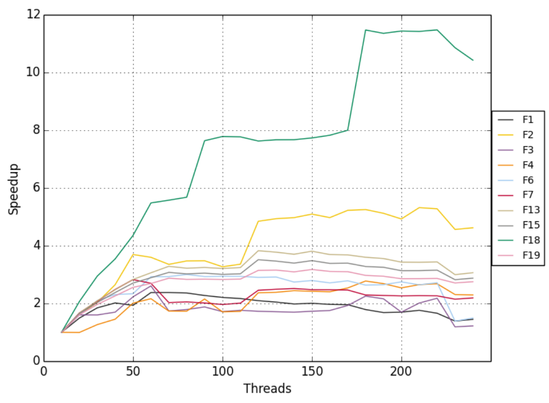


(a) Group B Speedup for Xeon Phi

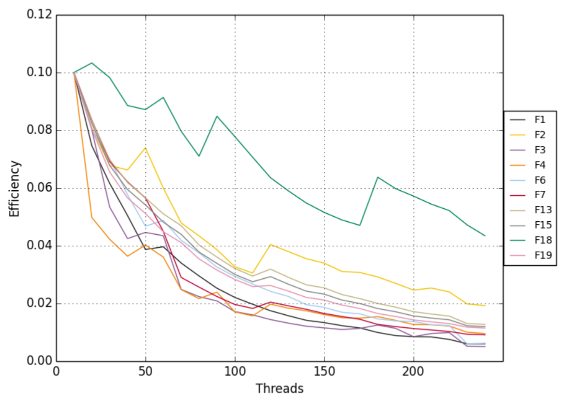


(b) Group B Efficiency for Xeon Phi

Fig. 10: Group B test functions for $N_d = 20$

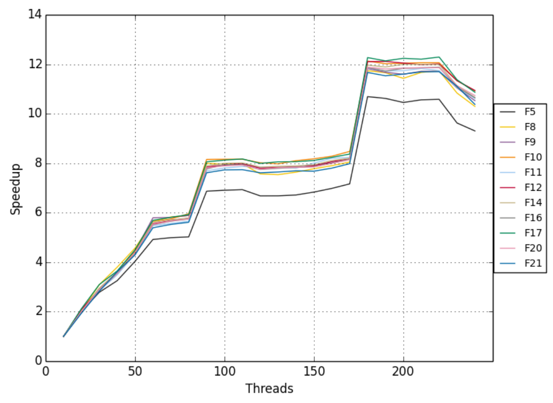


(a) Group A Speedup for Xeon Phi

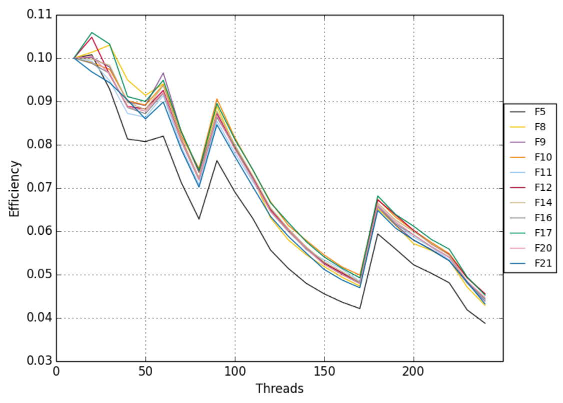


(b) Group A Efficiency for Xeon Phi

Fig. 11: Group A test functions for $N_d = 30$



(a) Group B Speedup for Xeon Phi



(b) Group B Efficiency for Xeon Phi

Fig. 12: Group B test functions for $Nd = 30$

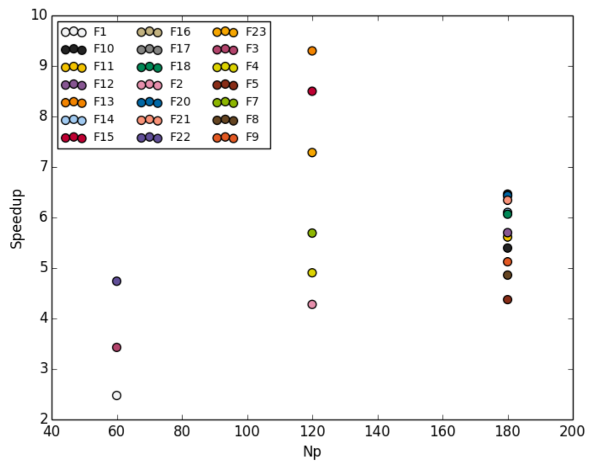


Fig. 13: N_p vs Speedup Thread Per Probe

between N_p and speedup, with the exception of the functions $F13$, $F15$ and $F23$, all of which are in group A.

V. Conclusion and Future Work

This study has shown that all test functions studied exhibited near linear, and sometimes super linear speedup on the host system with highs of 20 speedup for a large portion of group A test functions, and a maximum speedup of 35 for one test function. For both groups on the host system, speedup spiked when the thread count was equal to physical cores, but plateaued when the system was given more threads. This suggests that the algorithmic performance can be increased using OpenMP parallelization, though the thread count should never be greater than the physical cores available.

The native implementation for the Intel Xeon Phi architecture exhibited sub-linear speedup, but achieved up to 10 times speedup as compared to serial execution on the host system. The author added insights into the relationship between N_d, N_p and speedup by comparing groups A and B over both architectures, and found that thread subscription may have an affect on performance of higher dimensions. Both architectures performed faster per thread given higher dimensional decision space as well as N_p, and the Xeon Phi architecture implementation has shown to have room for future work.

Although the benchmarks on the host system show that this parallel implementation is capable of linear and at times super linear speedup, there is further work that will contribute to the growing literature on CFO including the implementation of CFO using other frameworks like Cilk, Threaded Building Blocks, or MPI and comparing CFO with variants of other parallelized population-based metaheuristic algorithms.

References

[1] R. A. Formato, "Central force optimization: A new metaheuristic with applications in applied electromagnetics," *Progress in Electromagnetics Research, PIER 77*, pp. 425–491, 2007. [Online]. Available: http://www.jpier.org/PIER/pier.php?paper=07082403

[2] R. Green, L. Wang, and M. Alam, "Training neural networks using Central Force Optimization and Particle Swarm Optimization: Insights and comparisons," *Expert Systems with Applications*, vol. 39, pp. 555–563, January 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0957417411010086

[3] R. Green, L. Wang, M. Alam, and R. Formato, "Central Force Optimization on a GPU: A case study in high performance metaheuristics using multiple topologies," in *IEEE Congress on Evolutionary Computation*, New Orleans, Los Angeles, June 2011, pp. 550–557. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5949667

[4] R. S. Sinha and S. Singh, "Optimization Techniques on GPU: A Survey," in *International Multi Track Conference on Science, Engineering & Technical Innovations*, Jalandar, India, June 2014.

[5] R. Green, L. Wang, M. Alam, and R. A. Formato, "Central force optimization on a GPU: A case study in high performance metaheuristics," *Journal of Supercomputing*, vol. 62, pp. 378–398, October 2012.

[6] S. Singh, J. Kaur, and R. Sinha, "A Comprehensive Survey on Various Evolutionary Algorithms on GPU," in *International Conference on Communication, Computing and Systems*, Ferozepur, Punjab, India, August 2014.

[7] E. Ahmed, K. R. Mahmoud, S. Hamad, and Z. T. Fayed, "CFO Parallel Implementation on GPU for Adaptive Beam-forming Applications," *International Journal of Computer Applications*, vol. 70, no. 12, pp. 10–16, May 2013.

[8] R. A. Formato, “Central Force Optimization: A New Nature Inspired Computational Framework for Multidimensional Search and Optimization,” in *NICSO*. Springer-Verlag, 2007, pp. 221–238. [Online]. Available: http://www.springerlink.com/content/t357063336g229g0/

[9] ——, “Pseudorandomness in Central Force Optimization,” *Computing Research Repository*, vol. abs/1001.0317, 2010. [Online]. Available: http://arxiv.org/abs/1001.0317

[10] ——, “Central Force Optimization with variable initial probes and adaptive decision space,” *Applied Mathematics and Computation*, vol. 217, no. 21, pp. 8866–8872, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0096300311005315

[11] ——, “Parameter-Free Deterministic Global Search with Central Force Optimization,” *Computing Research Repository*, vol. abs/1003.1039, 2010. [Online]. Available: http://www.springerlink.com/content/d45u6135702015wq/

[12] M. Chao, S. Z. Xin, and L. S. Min, “Neural network ensembles based on copula methods and Distributed Multiobjective Central Force Optimization algorithm ,” *Engineering Applications of Artificial Intelligence* , vol. 32, pp. 203–212, 2014.

[13] N. F. Shaikh and D. D. Doye, “An Adaptive Central Force Optimization (ACFO) and Feed Forward Back Propagation Neural Network (FFBNN) based iris recognition system ,” *Journal of Intelligence and Fuzzy Systems*, vol. 30, no. 4, pp. 2083–2094, March 2016.

[14] R. A. Formato, “Improved CFO Algorithm for Antenna Optimization,” *Progress in Electromagnetics Research, PIER B*, vol. 19, pp. 405–425, 2010. [Online]. Available: http://www.jpier.org/PIERB/pier.php?paper=09112309

[15] ——, “Central Force Optimization Applied to the PBM Suite of Antenna Benchmarks,” *Computing Research Repository*, vol. abs/1003.0221, 2010. [Online]. Available: http://arxiv.org/abs/1003.0221

[16] A. Haghighi and H. M. Ramos, “Detection of Leakage Freshwater and Friction Factor Calibration in Drinking Networks Using Central Force Optimization,” *Water Resources Management*, vol. 26, no. 8, pp. 2347–2363, March 2012. [Online]. Available: http://rd.springer.com/article/10.1007/s11269-012-0020-6

[17] A. Jabbary, H. T. Podeh, H. Younesi, and A. H. Haghiabi, “Development of central force optimization for pipe-sizing of water distribution networks,” *Water Science and Technology: Water Supply*, vol. 16, no. 5, pp. 1398–1409, 2016.

[18] S. M. J. Moghaddas and H. M. Samani, “Application of central force optimization method to design transient protection devices for water transmission pipelines,” *Modern Applied Science*, vol. 11, no. 3, p. 76, 2016.

[19] B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1999.