Global Minimum for Benchmark Optimisation Problems

Thomas Hodgson, Conor Osborne, Jonathan Spence

April 5, 2021

1 Introduction

The problem of minimising a function $f : \mathbb{R}^d \to \mathbb{R}$ appears in many areas of applied mathematics. Typically f is non-convex and traditional gradient based methods must be modified to sufficiently explore the space in order to find the global, opposed to a local, minimum. In many cases, e.g. the training of deep neural networks, the dimension of the problem can become extremely high leading to extremely large domains which are costly to explore. In this report, we consider the use of High Performance Computing (HPC) applied to several optimisation methods in order to speed up the exploration of the space. We present scaling analysis for the benchmark problems described in Section 1.1, where the dimensionality d remains low enough for the problem to be solved by a single processor.

Specifically, we consider the problem of finding

$$x^\star \in \arg\min_{x \in D} f(x),$$

where $D \subseteq \mathbb{R}^d$. We assume throughout that such $x^\star$ exists. Denote the optimal function value by $f^\star = f(x^\star)$. In this report we seek to approximate $x^\star, f^\star$ by $\hat{x}$ and $\hat{f} = f(\hat{x})$, respectively.

All the code for this project is available on GitHub at
https://github.com/Tom271/HPC4M_Optimisation.

1.1 Test Functions

We use the following functions as benchmark problems to test our optimisation methods:

- The Eggholder function function for $d = 2$:

$$f(x) = -(x_1 + 47)\sin\left(\sqrt{\left|\frac{x_0}{2} + x_1 + 47\right|}\right) - x_0 \sin\left(\sqrt{|x_0 - x_1 - 47|}\right),$$

for $x = (x_1, x_2) \in D = [-512, 512]^2$.

- The Shekel function with $d = 4$:

$$f(x) = -\sum_{i=1}^{10} \left(\sum_{j=1}^{4} (x_j - C_{ji})^2 + \beta_i \right)^{-1},$$

where

$$\beta = \frac{1}{10}(1, 2, 2, 4, 4, 6, 3, 7, 5, 5)^T$$

$$C = \begin{pmatrix} 4.0 & 1.0 & 8.0 & 6.0 & 3.0 & 2.0 & 5.0 & 8.0 & 6.0 & 7.0 \\ 4.0 & 1.0 & 8.0 & 6.0 & 7.0 & 9.0 & 3.0 & 1.0 & 2.0 & 3.6 \\ 4.0 & 1.0 & 8.0 & 6.0 & 3.0 & 2.0 & 5.0 & 8.0 & 6.0 & 7.0 \\ 4.0 & 1.0 & 8.0 & 6.0 & 7.0 & 9.0 & 3.0 & 1.0 & 2.0 & 3.6 \end{pmatrix},$$

and $x = (x_1, x_2, x_3, x_4) \in [0, 10]^4$.

For both problems we impose the approximations remain inside D by orthogonal projection of all points outwith D back onto the boundary of D.

2 Optimisation Methods

2.1 Black-box Methods

2.1.1 Grid Search

One of the most simple approaches to numerical optimisation is to perform a grid search. Here, the domain D is approximated by a finite collection of points $\mathcal{G} \subset D$. We can then approximate the global minimum of f by

$$\hat{x} = \arg\min_{x \in \mathcal{G}} f(x).$$

This approach is simple to implement, easy to parallelise and requires no very little assumptions on f. However, we demonstrate below that a simple grid search can be extremely costly:

Suppose that f is L-Lipschitz continuous on $D = [0, 1]^d$ and we choose the grid

$$\mathcal{G} = \left\{ \left(\frac{i_0}{N}, \cdots, \frac{i_d}{N} \right) \middle| \, 0 \leq i_j \leq N \text{ for } 1 \leq j \leq d \right\}.$$

Then, one can show that due to the Lipschitz continuity of f

$$\|f^\star - \hat{f}\| \leq \frac{L\sqrt{d}}{2N}.$$

Even in $d = 4$ dimensions, if we desire an error of at most 0.0001 in $\hat{f}$ then we require $N = 10000$, and $|\mathcal{G}| = (N + 1)^d \approx 10^{16}$ function evaluations. This is a fairly costly computation - even for a relatively simple problem. By increasing d only slightly, the computation time quickly becomes infeasible, with or without HPC, thus motivating the development of further methods.

2.1.2 Particle Swarm Optimisation

Particle Swarm Optimisation (PSO) [1] attempts to reduce the number of function evaluations needed to reach the global minimum using inspiration from swarm intelligence observed in nature. The method relies on a random initialisation of N particles $x_{n,0}$ in D, with random initial velocities $v_{n,0}$, for $1 \leq n \leq N$. Here, we use the notation $x_{n,t}, v_{n,t}$ to denote the position and velocity of particle n at iteration t. The particles each track their own personal best point,

$$x_n^{\mathrm{PB}} = \arg \min_{x \in \mathcal{H}(x_n)} f(x),$$

where $\mathcal{H}(x_n)$ denotes the history of particle n, as well as the current global best point

$$x_t^{\mathrm{GB}} = \arg \min_{1 \leq n \leq N} f(x_{n,t}^{\mathrm{PB}}).$$

At iteration t we take

$$x_{n,t+1} = x_{n,t} + v_{n,t},$$

before updating its personal and the global best points (if necessary). Each particle should then accelerate by random components in the direction of its personal best and the overall global best. Accordingly, we set

$$v_{n,t+1} = \gamma_0 v_{n,t} + \gamma_1 r_1 (x_n^{\mathrm{PB}} - x_{n,t+1}) + \gamma_2 r_2 (x_t^{\mathrm{GB}} - x_{n,t+1}),$$

for $r_1, r_2 \overset{\text{i.i.d}}{\sim} \mathrm{Uniform}[0,1]$ and numbers $\gamma_0, \gamma_1, \gamma_2 > 0$ which may depend on t.

Since the particles in the 'swarm' are constantly accelerating toward their own personal best and the global best, x_t^{GB} is able to quickly move close to $x^\star$ in many practice problems. In [1], it is also mentioned that it can be better to divide the N particles into smaller 'packs' of M particles, each pack with its own global best. This approach helps to avoid the swarm becoming stuck in a sub-optimal local minimum. Moreover, this provides a trivial way of dividing the N particles among n_p processors by taking $M = N/n_p$ and allowing each processor to perform an independent swarm using M particles. In general, a larger total number of particles allows a better exploration of the space (the number of particles required will increase with the size of the domain) and splitting the swarm into more packs makes the algorithm converge slower, but with less probability of becoming stuck in a sub-optimal minimum.

However, there is no exit condition for halting the swarm to ensure the global best is within a given distance of a minimum of f. Such criteria do exist for the gradient based methods considered in Section 2.2. Thus, it is sensible to use PSO as a means of exploring the domain D, and then continuing with a gradient-based approach starting at the final value of x_t^{GB}.

2.2 Gradient-Based Methods

Gradient based methods are widely used in optimisation, particularly for convex problems. Starting with an initial point $x \in D$ they iteratively search for a minimum of the objective function f with the direction of search defined by the gradient of f. All gradient based methods assume that the function f is differentiable. Here we look at two methods used to define the direction of search. For the gradient descent algorithm [2, Section 8.1] the direction of search is chosen to be the direction of steepest descent at each point. Whereas in the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm [3, Section 3.5] information about the curvature information, in the form of an approximation of the Hessian, is additionally considered to aid convergence.

2.2.1 Gradient Descent

If x_i is the candidate solution at stage i of the algorithm and then a natural choice for the search direction is the negative of the gradient $d_i = -\nabla_x f(x_i)$, which gives the direction of steepest descent at each point. For the updating rule we select a step size γ, then at stage i:

1. Update d_i via
$$d_i = \gamma \nabla_x f(x_i);$$
2. Update the sequence of candidate solution
$$x_{i+1} = x_i - d_i.$$

The algorithm continues until one of the following three conditions are met; the maximum number of iterations are exceeded, the norm of the gradient $\|\nabla f\|$ is less than the prescribed gradient tolerance or he relative change between x_{i+1} and x_i is less than the prescribed tolerance.

We are not constrained to picking d^i in this form. There are several methods that speed up the convergence of gradient descent including, momentum and Nesterov accelerated gradient. Please see the code repository for examples of these in action.

Gradient descent is guaranteed to converge provided the objective function f is Lipschitz, making it an effective for convex optimisation. Unfortunately, for non-convex f there is no guarantee of convergence to a global minimum. The algorithm will only find the global minimum if the initial point happens to be within the basin of attraction of the global minimum. To overcome this issue we look at starting multiple initial points, running each of them through a gradient descent algorithm, and then selecting the minimum from these minimum. If enough initial points are used we can be confident that one will find the global minimum. What is mean by enough will depend on the function we are minimising and the domain we are minimising it over.

Suppose we start with N particles, with each one performing a gradient descent algorithm. Then, as with PSO, a trivial way to parallelise this is to split the N particles over n_P processors, allocating $M = N/n_p$ to each.

2.2.2 BFGS

While Gradient descent is guaranteed to converge to a local minimum, the convergence rate is dictated by choice of the stepsize γ which is heavily problem specific and requires some fine-tuning to get right. An alternative approach is given by Newton's method [3, Section 3.5] which incorporates the curvature of the objective function f:

$$x_{i+1} = x_i - (\nabla^2 f(x_i))^{-1} \nabla f(x_i),$$

where $\nabla^2 f(x_i)$ is the Hessian matrix of f. However, especially in large dimensions, inversion of the Hessian matrix is costly. An alternative approach is to iteratively approximate $(\nabla^2 f(x_i))^{-1}$ at each timestep through the approximate equality

$$x_{i+1} - x_i \approx (\nabla^2 f(x_{i+1}))^{-1})(\nabla f(x_{i+1}) - \nabla f(x_i)).$$

Thus, let G_i be the approximation of $(\nabla^2 f(x_i))^{-1}$ at iteration i and define $s_i := x_{i+1} - x_i$, $q_i := \nabla f(x_{i+1}) - \nabla f(x_i)$. We iteratively construct G by enforcing it to satisfy the relation above, that is

$$s_i = G_{i+1} q_i. \tag{1}$$

Moreover, to deduce G_{i+1} from G_i we enforce

$$G_{i+1} = G_i + E_i, \tag{2}$$

where E_i is symmetric and positive-definite (this condition ensures $-G_i \nabla f(x_i)$ is a descent direction, provided G_0 is symmetric and positive-definite). The descent update is then given by

$$x_{i+1} = x_i - \gamma G_i \nabla f(x_i).$$

One of the most successful algorithms of this type is the BFGS iteration [3, Section 5.5.2] which updates G according to (1), (2), where

$$E_i = \frac{s_i s_i^T}{s_i^T q_i} - \frac{G_i q_i q_i^T G_i}{q_i^T G_i q_i} + w_i w_i^T,$$
$$w_i = \sqrt{q_i^T G_i q_i}\left(\frac{s_i}{s_i^T q_i} - \frac{G_i q_i}{q_i^T G_i q_i}\right).$$

In practice, the BFGS iteration is often able to converge faster than a standard gradient descent. However, BFGS also suffers the issue of becoming stuck in a local, sub-optimal, minimum. It therefore becomes necessary to employ many runs of BFGS to allow exploration of the entire state space as with gradient descent. Parallelisation then occurs in a similar fashion to the gradient descent method. In our code, we use (serial) BFGS to obtain a fast descent towards the minimum preceded by a (parallel) particle swarm optimisation which seeks to move close to the global minimum. The fast convergence of BFGS when near a minimum means this serial section of code doesn't have a significant impact on parallel performance.

3 Parallel Architecture

3.1 Gradient-Based Methods

As discussed in Section 2.2, the gradient based methods guarantee convergence. However, this convergence may be to a local suboptimal minimum rather than the global minimum. To explore the whole domain we must set off many runs of gradient descent or BFGS and hope that one of these runs converges to the global minimum.

For each method we pick N particles that are randomly distributed across the domain D where N is chosen sufficiently large for the global minimum to be found most of the time.

For n_p processors we parallelise this by sending $M = N/(n_p - 1)$ particles to each processor worker process, that is all processors except the root, which is left as as the communicator between all processors. Each worker processor then runs gradient descent or BFGS for the M particles and returns the minimum found from those M to the root process. The root process then returns the lowest of the local minimum it has received.

The only communication that occurs is when each worker process returns the minimum it has found the root process. While this is highly efficient in terms of parallelisation it is inconvenient for analysis of the speed of convergence. The root process cannot return the overall minimum until all worker processors have returned their individual minimums. So the time taken to find the minimum will be limited by the time taken by the final processor to send back its minimum.

Hence, for speed analysis we alter the algorithm so that as soon as one of the N particles is within 10^{-4} of the true global minimum the algorithm is ended for each processor and the global minimum is returned. Whilst this is obviously infeasible if we do not know the true minimum it is an opportunity for us to show that for the toy problems in Section 1.1 that with more processors gradient descent is faster at finding the global minimum.

In practice each worker process communicates with the root process periodically, every τ iterations, inform if it has found the global minimum and to find out if one of the other processors has reached the global minimum. There is clearly a trade off here, if τ is small then each worker process learns that another process has found the global minimum in fewer iterations but the need to communicate more often causing a bottleneck. Alternatively with a larger τ we have less communication slow down but a process might run a large number of iterations unnecessarily after the global minimum has been found elsewhere.

3.2 Particle Swarm Optimisation

As discussed in Section 2.1.2, it is possible to perform particle swarm optimisation by splitting the swarm into 'packs' which do not communicate until the termination of the algorithm. This provides a very useful way to parallelise PSO optimisation without requiring all processes to communicate at each other to determine the global best point between all processors. Having this level

of communication for many iterations will severely impact the parallel code, especially when the dimensionality of the problem, and thus the size of each communication, becomes large. Instead, each processor can perform a single, independent PSO with its own global minimum. The only communication is then to compare end results once each processor has completed their PSO.

Assume we wish to use n_p processors to solve the problem. If we wish to explore the space using N particles, we can use strong scaling by allowing each processor to use $M = \lfloor N/n_p \rfloor$ particles in its own swarm, before collecting the minimum result in a root process. This reduces the problem size for each process, decreasing the runtime. If n_p does not divide N, we can solve the discrepancy in problem sizes by adding a single extra particle to exactly $N/n_p (\text{mod } n_p)$ processors. Provided M is sufficiently large, this extra particle will not severely impact the runtime of the swarm. On the other hand, one can achieve strong scaling by fixing the number M particles to assign to each processor and then using a total of $N = n_p M$ particles in the swarm. Since the problem size assigned to each process remains the same, the runtime should remain constant regardless of n_p. However, as n_p increases we use a larger swarm allowing a better exploration of the space.

Recall that the PSO does not have an exit criteria for a given error tolerance. To achieve a desired error, one could follow up a particle swarm with a gradient-based method. This induces a serial component to the code. However, typically sufficiently large swarms will return very good approximations of $x^\star$. Thus, the gradient method will require only a few iterations to converge, keeping the size of this serial component small.

4 Scaling Analysis

We present scaling analysis for the parallel methods discussed in Section 3 applied to the benchmark problems detailed in Section 1.1. All experiments are ran on a single compute node in the cirrus computing cluster (http://www.cirrus.ac.uk).

4.1 Gradient-Based Methods

We apply the gradient descent algorithm discussed in Section 3.1 to the benchmark problems given in Section 1.1. N will vary depending on the particular function we are minimising and the domain it is being minimised over. For the eggholder function we use $N = 1000$ and for the shekel function we use $N = 500$. Note that this was selected experimentally by checking that the algorithms we use do indeed return the global minimum each time the code is run and a much smaller N may suffice. For the purpose of this essay however, where the objective is to compare parallelisation methods, this approach is adequate. We set

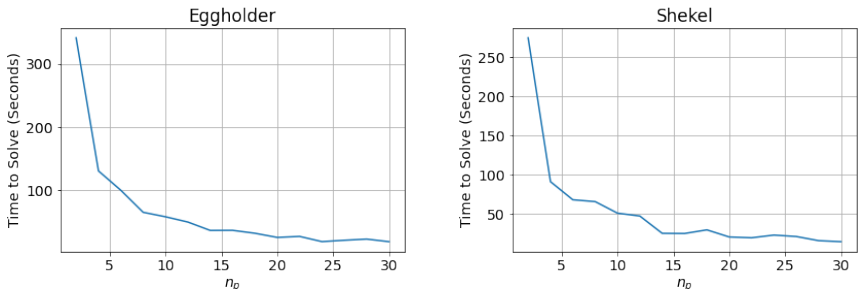

Figure 1: Computation time of gradient descent method for benchmarch problems.

the period of communication $\tau = 1000$ and the time taken for the algorithm to complete is averaged over 10 runs for each n_p.

We only look at strong scaling in this case, that is where the overall size of the problem, i.e. N, stays constant when when n_p is increased. We avoid weak scaling, where N increases linearly with n_p, because the advantage of having a larger N comes in the form of being able to explore the space better. If N is too small then the global minimum may not be found at all, and if N is very large then many particles may end on the same local minimums, effectively meaning we are running a very large number of redundant gradient descents. Either way no reasonable analysis can be done.

The computation time for each the parallel gradient descent to converge to within 10^{-4} for each benchmark function can be see in Figure 1. The strong scaling speedup analysis for both functions can be see in in Figure 2. We can see that for both the eggholder and shekel function having more processors means that the local global minimum is found faster. Averaged over 10 runs the serial gradient descent took 341.2 and 271.6 seconds to find the minimiser for the eggholder function and shekel function respectively. The efficiency for both functions decreases immediately when n_p is increased. However, the observed strong speedup appears to be roughly linear for both functions indicating the efficiency loss from parallelisation increases linearly with n_p. Since communication only happens between the root processor and each worker processor the loss from communication is $O(n_p)$ rather than $O(n_p^2)$ if all processors were communicating with all other processors.

Unfortunately, the speed up for each function is not as stable as we might like. This is due to the natural of the initial points we are setting off. Since the initial points are randomised however this is not always the case. Ideally we would average over a much larger number of runs for each n_p but in the interest of not using up all the available computing hours we have limited ourselves to 10 runs for each n_p.

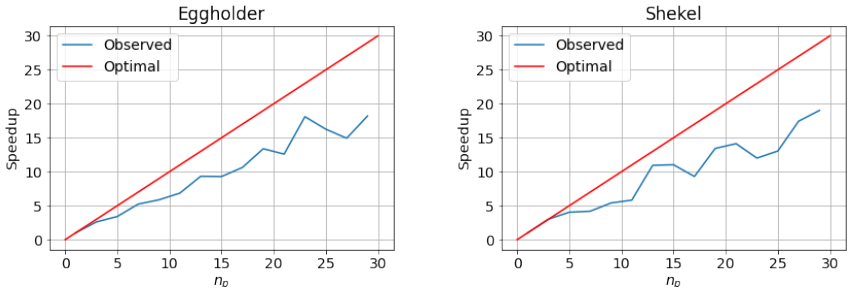

Figure 2: Speedup for strong-scaling of gradient descent method for benchmark problems.

4.2 Particle Swarm Optimisation with BFGS

We use the PSO method discussed in Section 3.2 applied to both test problems, where the swarm is followed by a BFGS descent until convergence is satisfied. For strong scaling, we consider a setup with 1000 iterations using both $N = 1000$ and $N = 100$ particles with $M \approx N/n_p$ particles assigned to each processor, for $1 \leq n_p \leq 30$ processors. For the benchmark problems, we reliably observe an error of 10^{-4} only when using $N = 1000$ particles, giving a more thorough exploration of the space, and the analysis for $N = 100$ particles instead provides an idea of how the scaling is affected by the size of the problem. The serial code is able to optimise the Eggholder function in 0.31 and 3.1 seconds and the Shekel function in 1.3 and 13 seconds for $N = 100$ and $N = 1000$ particles, respectively.

The speedup for each problem is given in Figure 3. The method displays optimal efficiency for small $n_p \leq 6$ in all cases. Eventually, the efficiency tails off for larger n_p since the communication overhead (due to having more processes) begins to dominate over the decreasing runtime of the individual swarms. For example, with $N = 100$ particles, the communication portion of the Shekel function solver occupies 0.08% and then 26% of the overall runtime when using $n_p = 2$ and 30 processors, respectively. The efficiency tails off at smaller n_p for $N = 100$ particles, since the overall runtime of this code is smaller and more quickly becomes dominated by the cost of communication. Moreover, the efficiency tails off more abruptly for the Shekel function - likely due to the fact that the problem in $d = 4$ dimensions requires a longer communication time than the Eggholder function in 2 dimensions. To highlight this, using $N = 100$ particles and 30 processors, the communication portion occupies 22% of the Eggholder optimisation, compared to 26% for the Shekel function.

Alternatively, we can consider weak scaling of PSO optimisation, assigning fixed amount M particles to each processor leading to $N = n_p M$ total particles. The runtime is recorded for various number of processors using $M = 100$ particles per process for both problems, and shown in Figure 4. In both cases we

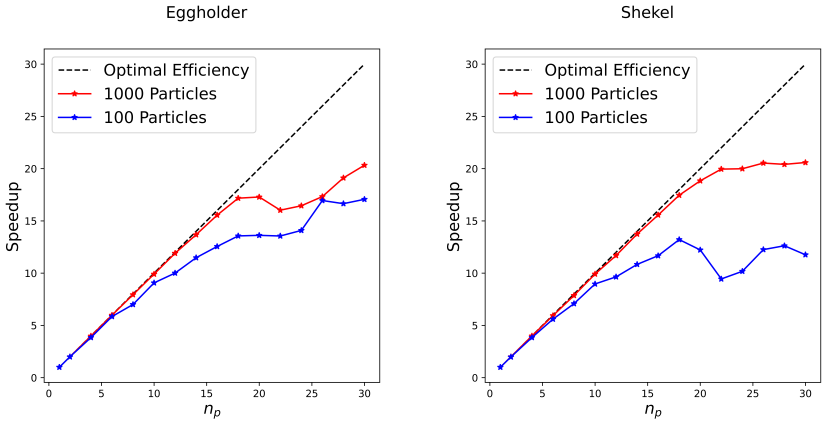

Figure 3: Speedup for strong-scaling of PSO method for benchmark problems.

observe optimal scaling (i.e. no increase in runtime, since the work assigned to each processor is independent of the number of processes) until around $n_p = 18$, after which there is a gradual increase in the runtime. As noted above, increasing the number of workers adds to the communication overhead which contributes to a slight increase in the runtime. Moreover, it is worth noting that compute nodes on cirrus are split into 2 processors with 18 cores. Hence, when using more than 18 processes we must send data between different processors at the end of the PSO, which is slightly more costly than sending data between processes on the same processor.

5 Conclusion

In this report, we looked into different ways of parallelising optimisation algorithms. We saw that while traditional gradient methods such as gradient descent and BFGS come with solid convergence guarantees; when tested on difficult, non-convex functions they require many runs from different initial points to find the global minimum. This renders such methods difficult to implement in parallel, since each particle converges at a different rate and lots of communication is required between processors to determine the optimal point. These hindrances provide sub-optimal speedup of the parallel architecture.

Conversely, particle swarm optimisation provides an effective way to explore the space with relatively few particles and is trivially parallelisable, but comes without the convergence guarantee of the gradient methods. It is then necessary to run the PSO of a given size for a set number of iterations, performed in parallel, before concluding with a single BFGS descent in serial. Since the bulk of the computation time is spent exploring the domain, we observe very near optimal speedup for this setup for a small enough number of processors.

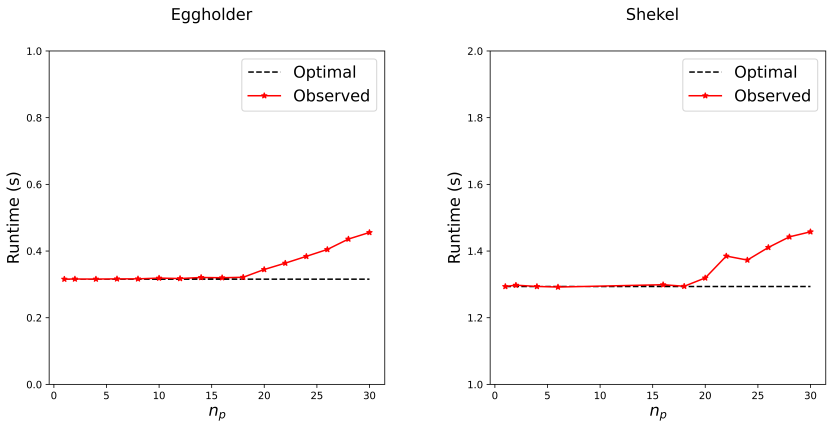

Figure 4: Runtime for both test problems using weak scaling of PSO with $M = 100$ particles per process.

The speedup increases with the number of particles required, which increases as the complexity of the problem increases, and the parallel architecture will thus improve PSO performance for larger-scale problems.

One possible avenue of future exploration on this subject is the integration of gradient methods into the particle swarm optimisation itself. Particularly, in such a way that enables the PSO to converge to the global minimum in parallel, without the need for a serial BFGS after the swarm, and such that the parallel speedup remains near-optimal.

Acknowledgements

This work used the Cirrus UK National Tier-2 HPC Service at EPCC (http://www.cirrus.ac.uk) funded by the University of Edinburgh and EPSRC (EP/P020267/1). TH was supported by The Maxwell Institute Graduate School in Analysis and its Applications, a Centre for Doctoral Training funded by the UK Engineering and Physical Sciences Research Council (grant EP/L016508/01), the Scottish Funding Council, Heriot-Watt University and the University of Edinburgh. JS and CO were supported by The Maxwell Institute Graduate School in Mathematical Modelling, Analysis and Computation, a Centre for Doctoral Training funded by the UK Engineering and Physical Sciences Research Council (grant EP/S023291/1), the Scottish Funding Council, Heriot-Watt University and the University of Edinburgh.

References

[1] R Eberhart and J Kennedy. A new optimizer using particle swarm theory. In *MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pages 39–43. IEEE, 1995.

[2] Amir Beck. *First-Order Methods in Optimization*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2017.

[3] Francisco J Aragón. *Nonlinear Optimization*. Springer Undergraduate Texts in Mathematics and Technology. Springer International Publishing : Imprint: Springer, Cham, 1st ed. 2019.. edition, 2019.