Applying Automated Optimisation Techniques to HPC Applications

T.D. Edwards
Cray Centre of Excellence for HECToR
Cray UK Ltd, Edinburgh, UK
tedwards@cray.com

Abstract—**Porting and optimising applications to a new processor architecture, a different compiler or the introduction of new features in the software or hardware environment can generate a large number of new parameters that have the potential to affect application performance. Vendors attempt to provide sensible defaults that perform well in general, for example grouping compiler optimisations into flag groupings and setting the default value of environment variables, they are inevitably based on the experience gained or expected behaviour of a normal application. In many cases applications will exhibit some behaviour that differs from the norm, for example requiring identical floating point results when changing MPI decompositions, or sending or receiving messages of unusual or irregular sizes. Manually finding the combination of flags and environment variables that provide optimum performance whilst maintaining a set of application specific criteria can be time consuming and tedious. There are a wide variety of potential algorithms and techniques that can be employed, each with various merits and suitability to the problem of optimising an HPC application. This paper explores, evaluates and compares techniques for automated optimisation HPC application parameters within fixed numbers of iterations.**

Index Terms—**Parameter Optimisation, Particle Swarm, Simulated Annealing, Curve Fitting, Interpolation, CUG 2012.**

I. Motivation

Achieving optimal performance is critical to High Performance Computing (HPC). The combined complexity of applications, system software and hardware present a wide variety of configurable settings that can affect runtime performance. The rapid pace and evolving nature of cutting edge technology and the demand for ever greater application performance means application settings have to be regularly revised to keep applications in peak condition.

As a result of this complexity it is difficult and time consuming to make predictions about how changing an individual setting will affect the overall performance of an application. Parameters may produce a non-linear and/or discontinuous response, or, in the case of application code, documentation may be inaccurate or incomplete. In other cases the application may have specific requirements, like *bit-reproducibility*[1], or numerical stability which place additional restrictions on the range and value of certain parameters. It may often be the case that a whole application has to be compiled with many optimisation disabled because of the side-effect some have on only a few files. Resolving this leads to vast search spaces which cannot be optimised using exhaustive search like those used in the Flamingo framework[1], or simple techniques like a binary search.

Instead, this paper evaluates some common heuristic optimisation algorithms to some typical example problems encountered by users of HPC.

II. Search Methods

An HPC application can be considered a function, $f(\mathbf{x})$, that maps a multi-dimensional parameter space (e.g. cache blocking parameters, MPI protocol boundaries, or individual compiler flags) to a scalar value to be minimised (typically the running time of the simulation). There are, however, no guarantees about the differentiability or continuity of $f(\mathbf{x})$ which eliminates use of standard analytical optimisation techniques. Also, each evaluation of $f(\mathbf{x})$ is potentially very expensive, limiting the number of evaluations and precluding exhaustive searches the entire parameter space. A selection of optimisation algorithms that do not require differentiable or continuous functions are described below:

Interval Search (IS): Perhaps the most common technique employed, the search space is sampled at regular intervals with equal distance between points. This approach can be very successful for smooth and shallow search spaces, however can be subject to aliasing if there are insufficient samples. To allow the measurement of statistical properties of what is essentially a static technique, a randomised offset is applied to the first point which causes the interval samples to be offset by different values each time the algorithm is started. As the simplest and most commonly used technique, its

[1]Bit-reproducibility is the requirement that the application will produce an identical result when run in different parallel decompositions. This places additional requirements on the application, system libraries, compiler and system hardware. It is a common requirement for applications used to study the natural environment.

performance is used as the benchmark from which to compare other techniques.

Random Search (RS): A naive stochastic search that randomly selects and evaluates points from the parameter space. Though this technique is very simple to implement, it uses no information from the search history, other than to avoid previously sampled points; each point is selected independently of all others.

Simulated Annealing (SA): Simulated Annealing is well established advanced textbook technique for optimisation[2]. The algorithm emulates the real-world process of liquid to solid annealing and the associated energy minimisation. The algorithm starts from a current configuration or position, $\mathbf{x}$, which has an associated energy, $e = f(\mathbf{x})$ (or time in the case of HPC optimisation). At each iteration a new configuration, $\mathbf{x}_{new}$, is randomly selected within distance, d, of $\mathbf{x}$ (using a standard Euclidean metric) and the new energy calculated, $e_{new} = f(\mathbf{x}_{new})$. The algorithm then accepts or rejects the new configuration over the previous one according to the Metropolis probability function:

$$P(e_{new}, e, T) = \exp\left(\frac{e - e_{new}}{T}\right)$$

This function depends upon differences in energy between the two states, and the current temperature T. If the temperature is zero then the algorithm will become greedy and only accept new states that have lower energy than the current state.

To simulate the cooling process, the value of T decreases with each iteration. This is calculated as an exponential decay from the initial maximum temperature T_{max} with a half-life of λ_T, (i.e. the temperature halves every λ_T iterations), the equations for the temperature at iteration i, T_i is:

$$T_i = T_{max}e^{ik},\ k = \frac{\ln(\frac{1}{2})}{\lambda_T}$$

Therefore this algorithm has three user tunable arguments that control the performance of the algorithm: d, T_{max} and λ_T.

Particle Swarm (PS): Just as Simulated Annealing is inspired by the real world processes in cooling materials, the Particle Swarm techniques are inspired by the real-world movements of flocks or swarms[3]. By storing information about previously encountered minima and communicating information between the members of the swarm the parameter space is explored with feedback from each particle.

N search particles are generated, with the ith particle having properties that are analogous to position, $\mathbf{x}^i$, and velocity, $\mathbf{v}^i$, both of which are initialised to random values. In addition, each particle also stores the position of the minimum value encountered, $\mathbf{x}^i_{min}$ and the position of the global minimum found by all the members of the swarm, $\mathbf{x}^i_g$. Each particle has its new velocity calculated in turn using the formula:

$$\mathbf{v}^i_{new} = \omega\mathbf{v}^i_{old} + \phi_l r_l(\mathbf{x}^i - \mathbf{x}^i_{min}) + \phi_g r_g(\mathbf{x}^i - \mathbf{x}^i_g)$$

where $r_{local}, r_{global} \in \mathcal{U}[0,1]$. The particle is then moved to its new position, $\mathbf{x}^i_{new}$, using a simple Euler step time-integration (with periodic boundary conditions):

$$\mathbf{x}^i_{new} = \mathbf{x}^i_{old} + \mathbf{x}^i_{new}$$

The values of the tunable arguments ω, ϕ_l,ϕ_g and the number of particles N will affect the performance of the algorithm.

Curve Fitting (CF): Though the response of an application to an input parameter could be potentially be highly sensitive or chaotic, in general it is not. In most cases there is a clear, if undetermined, relationship between the input parameter and the output energy (or run length). Therefore it is likely to be possible to interpolate between individual results and hopefully identify the global minimum in the parameter space.

To initialise this algorithm three random points are evaluated and used to construct an N dimensional interpolating fit using the Radial Basis Function (RBF) routine available in the SciPy[4] scientific software library. The resulting interpolation function is then used to quickly find the estimated global minimum using standard optimisation techniques for analytical functions. The predicted global minimum point is then used as the next evaluation point and process repeated with all points previously evaluated incorporated in the interpolation function. Algorithm 1 provides a pseudo-code implementation of the technique. To avoid being trapped by local minima, if the algorithm selects a previously evaluated point any required next point is selected at random. This provides the algorithm with the ability to escape local minimums and sample a the wider parameter space within the fixed number of iterations.

It is, however significantly more expensive computationally than other techniques due to the intensity of creating and evaluating the fitting solution. However, this cost is insignificant compared to the cost of evaluating, $f(\mathbf{x})$ which may be on the order of minutes or hours and require many cores.

As previously described algorithms, there are arguments which can affect the performance of the algorithm. Specifically, arguments affecting the quality of the fit like the choice of basis function and the amount of smoothing applied. Small experiments have shown that most successful solutions use either the multi-quadric and cubic spline basis functions and include some degree of smoothing. The smoothing value determines how far the interpolation is allowed to deviate from the know valued points, a value of 0 forces the interpolation function to pass through each real valued point. Tests will be conducted to determine the optimum basis function and value for the smoothing argument for general use.

Algorithm 1 Fortran style pseudo-code for the Curve Fitting Algorithm

```
! Initialise three random points
do i=1,3
  samples(i) = randomPoint()
end do

do i=4,max_iterations
  interpolationFunc = RBF(samples(1:i), smoothing)
  newpoint = find_minimum(interpolationFunc)
  if(newpoint in samples) then
     newpoint = randomPoint()
  samples(i) = newPoint
end do

return min(samples)
```

III. Example Search Spaces

Most real or integer valued search spaces can be qualitative described as being somewhere between smooth and shallow or sharp and steep. The exact nature of a particular search space is impossible to determine beforehand, therefore any potential search methodology must ideally be able to efficiently optimise search spaces at either extreme or at any point between. The search space may also be multidimensional if the result depends upon the value of more than one parameter.

To test the algorithms in Section II data was collected from an exhaustive search of four parameter spaces which exhibit the extremes of Smooth and Shallow to Sharp and Steep in both one and two dimensions. Though these search spaces were small enough to be analysed by exhaustive search, they do exhibit behaviour which is typical of the types of problems encountered in HPC and have the advantage of being measurements on real hardware.

- *1D Smooth and Shallow* - The results of sweeping over a single cache blocking parameter. The space can be described as smooth and shallow with a clear global minimum once measurement noise is accounted for. The global minimum is 8.3% faster than the mean value of the space. (Figure 1)
- *1D Sharp and Steep* - The results of sweeping the same cache parameter as the first space, but this time the OpenMP parallel region is active with four threads. The space becomes sharper and steeper with at least five identifiable local minima. The global minimum is 32.0% faster than the mean value. (Figure 2)
- *2D Smooth and Shallow* - A sweep over two parameters controlling the cache blocking of a simple matrix-multiplication. The optimum results will typically depend upon the properties of the processor being used and features like the cache architecture. Even though the space can be described as smooth and shallow, the global minimum is 44.4% faster than the mean value, a significant potential speed up over the expectation value. (Figure 3)
- *2D Sharp and Steep* - The Cartesian product of two 1D parameters with similar features as the 1D Sharp and Steep dataset. Two 1D Sharp and Steep datasets were added combined to produce this 2D sharp and steep dataset. The global minimum is 32.2% faster than the mean value. (Figure 4)

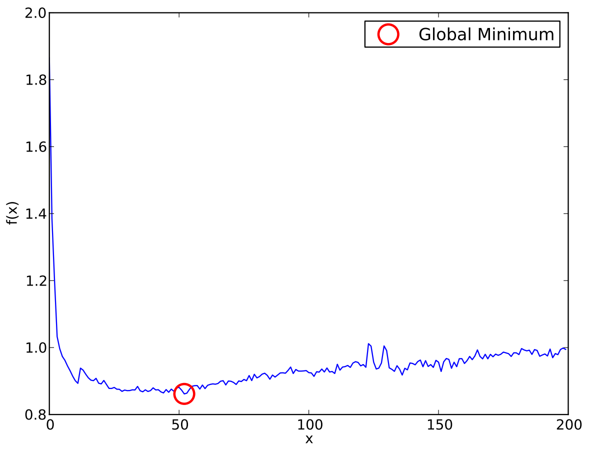

Figure 1: 1D Smooth and Shallow Dataset

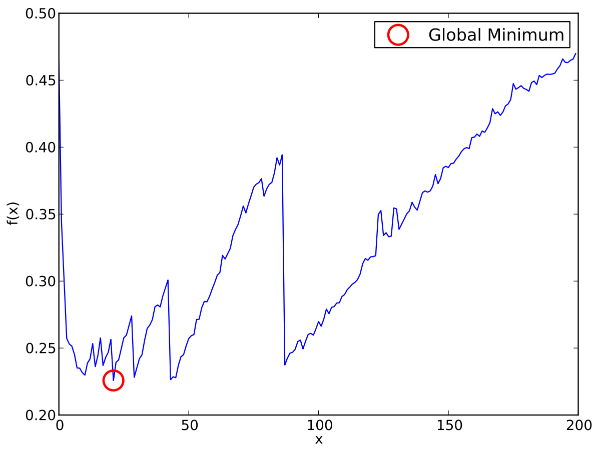

Figure 2: 1D Sharp and Steep Dataset

IV. Evaluation Method

In production situations there is no method of determining whether the minimum value found is the global or just a local minimum without having explored the entire parameter space. As most experiments are limited by the number of evaluations that can be run with the resources available the priority is finding the algorithm and arguments that find the best solution in a fixed number of iterations, rather than the algorithm

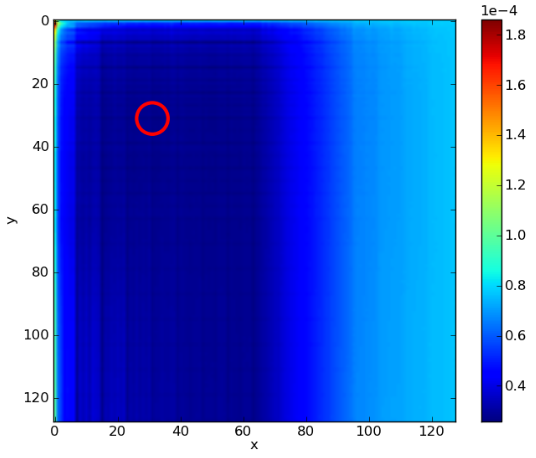

Figure 3: 2D Smooth and Shallow Dataset

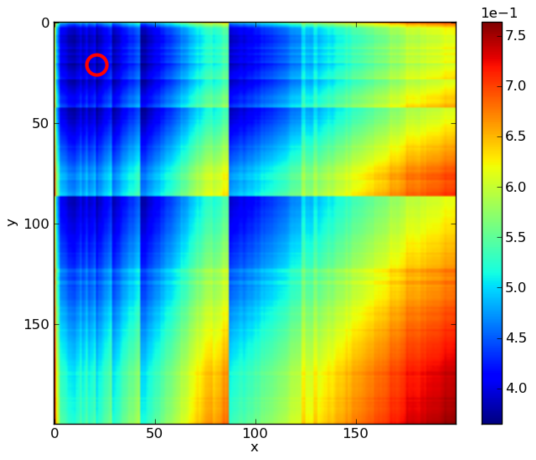

Figure 4: 2D Sharp and Steep Dataset

that converges in the fewest number (though these may be one and the same).

Each algorithm's performance was evaluated by running 1000 experiments with fixed numbers of iterations on each dataset. Iterations were limited to 4, 8, 16, 32 and 64. Each experiment returned the optimum value that it found, f_{best}, which was converted to a relative solution value, r, by taking the global maximum, $f_{\max}$, and the global minimum, $f_{\min}$, values for the dataset and applying the following equation:

$$r = \frac{f_{\text{best}} - f_{\min}}{f_{\max} - f_{\min}}$$

This results was then averaged to find the *mean relative solution*, the value that might be expected to be found for this algorithm when applied to this problem. This value can then be used to fairly compare different algorithms; a lower value implying the algorithm was more efficient and more likely to find a better optimum value.

V. Results

Simulated Annealing, Particle Swarm and Fitting search, each take additional arguments which affect their performance. Finding the optimum values is a separate optimisation problem and can be be performed recursively by applying the algorithms to their own optimisation. For the requirements of this paper, however, the values should have good (if not optimal) performance properties for as wide a variety of data sets and algorithms as possible.

Algorithm	Parameter Settings
Simulated Annealing	$T_{max} = 0.25$, $d = 0.5$, $\lambda_T = 2.0$
Particle Swarm	$N = 2$, $\omega = 0.8$, $\phi_l =$, $\phi_g = 0.0$
Fitting Search	Basis function = *Multi-quadric*, Smoothing=0.02

Table I: Optimised algorithm arguments for continuous parameter search

Table I shows the arguments found to provide the best results across each of the search spaces iteration count limits tested. The results from experiments with these argument values, being the best for the individual algorithms, are then used to compare performance across algorithms.

Full results for each algorithm and data space are recorded in Table II and illustrated in Figures 5 and 6

Iters	IS	RS	SA	PS	CF
4	*1.73*	2.80	4.07	2.77	2.29
8	*1.05*	1.57	2.46	1.48	1.62
16	0.720	0.891	1.12	0.894	*0.622*
32	0.472	0.549	0.518	0.559	*0.334*
64	0.256	0.297	*0.215*	0.342	0.226

(a) 1D Smooth and Shallow

Iters	IS	RS	SA	PS	CF
4	*8.46*	14.2	19.1	13.8	13.9
8	*5.69*	7.48	10.4	7.16	6.56
16	3.31	3.88	5.53	4.00	*2.49*
32	1.04	1.99	2.50	2.05	*0.841*
64	0.627	0.775	0.691	1.02	*0.349*

(b) 1D Sharp and Steep

Iters	IS	RS	SA	PS	CF
4	3.45	3.32	5.37	*3.24*	3.30
8	2.97	2.35	2.73	2.22	*1.65*
16	2.02	1.47	1.37	1.43	*1.05*
32	1.59	0.925	*0.861*	0.909	0.944
64	1.26	0.590	0.537	0.623	*0.391*

(c) 2D Smooth and Shallow

Iters	IS	RS	SA	PS	CF
4	*20.1*	21.9	24.2	21.1	21.7
8	15.8	15.0	16.7	*14.3*	15.6
16	*9.89*	10.4	11.7	10.7	10.8
32	7.44	7.50	7.55	7.48	*6.27*
64	5.97	5.62	5.36	5.72	*3.93*

(d) 2D Sharp and Steep

Table II: The mean relative solution ($\times 10^{-2}$) for each algorithm and search space

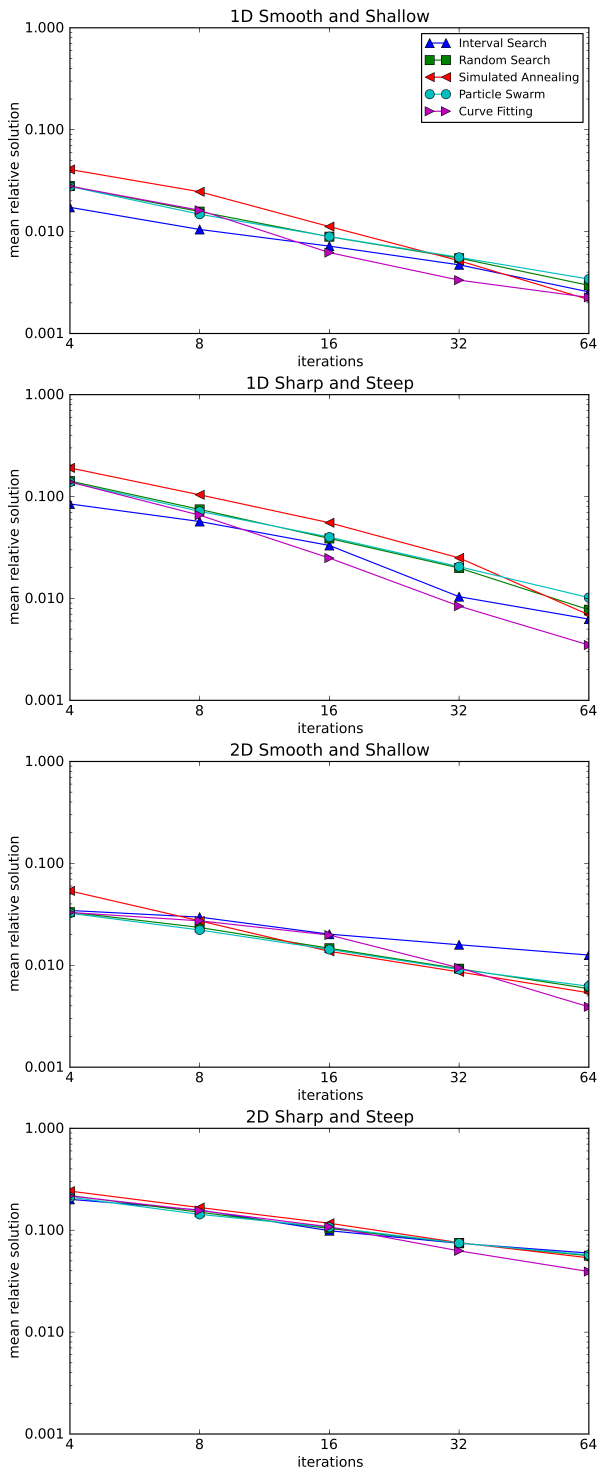

Figure 5: Mean relative solution of each algorithm on each problem

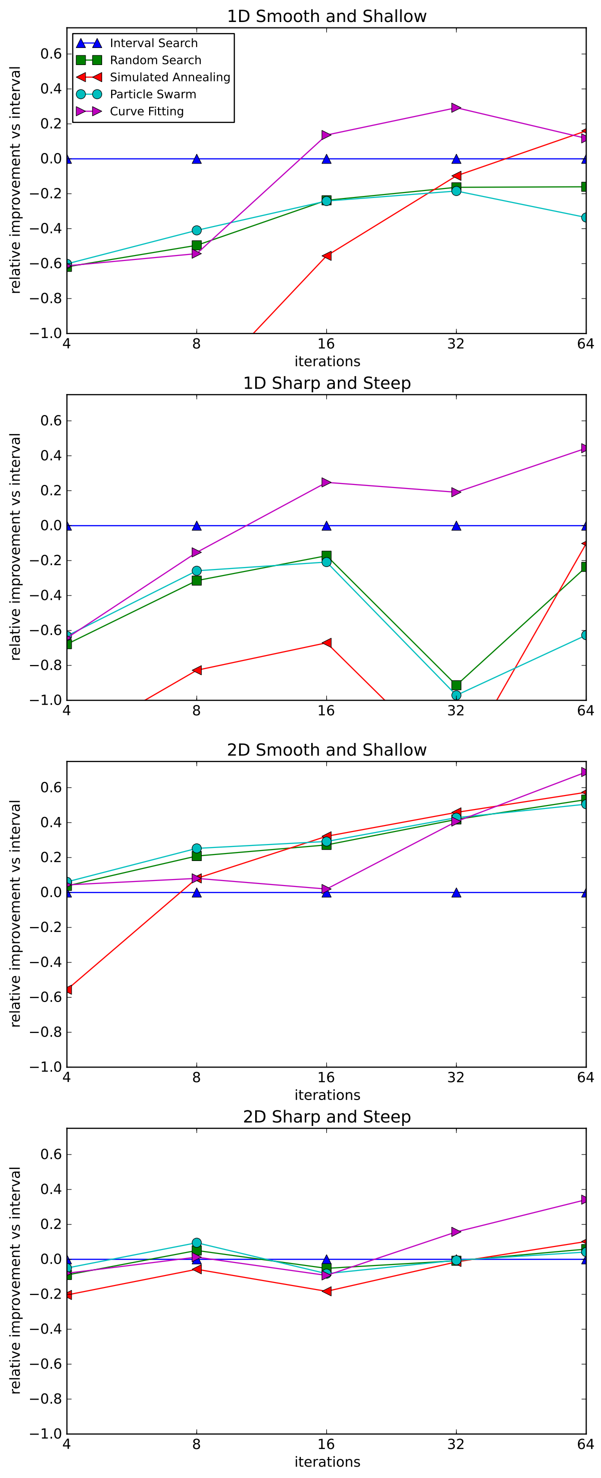

Figure 6: Relative improvement of each algorithm vs interval search on each problem.

VI. Conclusions

From this study it is possible to conclude, for these search spaces and with the algorithms tested, that:

- Performing more iterations produces consistently better results with all algorithms, though in all cases the relative improvement diminishes as increasingly more iterations are applied.
- Smooth and Shallow search spaces are optimised in fewer iterations than Sharp and Steep ones - There is almost an order of magnitude difference between the Sharp and Steep spaces versus the Smooth and Shallow for each corresponding algorithm and iteration limit.
- Optimising in multiple dimensions simultaneously is less efficient than optimising independent parameters separately. Optimising the 2D Sharp and Steep space as independent arguments rather than as the combined 2D space would have produced a significantly better result in the same number or fewer iterations (using 1D Sharp and Steep as a reference).
- Particle Swarm and Simulated Annealing do appear to be efficient methods for these search spaces with the limited number of iterations. In most cases they do not produce results significantly better than Random Search. This is most likely due to the limit on the number of iterations of the algorithm.
- When performing very few iterations (e.g. 4) interval search performs best however for counts between 8 and 16 using a standard Random Search method produces better results. The Curve Fitting technique provides excellent results on the larger iterations counts (32 and 64), producing a means solution that significantly better than any other for the more compilcated search spaces. The Curve Fitting technique should be considered, when large numbers of iterations are possible (evaluations are cheap and quick), the search space is very large (e.g. it spans more than one dimension), or when the parameter has a signficant impact on performance.

VII. Application to Optimising Compiler Flags

Motivation

An extension and frequently encountered problem is searching for the optimal combinations of compiler flags, especially when there are additional complications like maintaining a executable's bit-reproducibility or numerical stability. Certain combinations of compiler optimisations have the potential to adversely affect the numerical properties of calculations. This, combined with the fact that the majority of applications only choose a single set of flags to be used by the compiler, mean applications may be compiled without beneficial optimisation included.

Ideally it would be possible to identify the optimal flags for each file while still having the required numerical properties. In the simplest cases, where only a single file requires modification, binary search can be used to identify the file requiring different settings, however it is typically impossible to know beforehand how many files require modification. Binary search is also limited to a choice of only two sets of compiler options. However, this problem has much in common with the well-studied travelling salesman problem, to which Simulated Annealing has been very successfully applied[2], so the same is tried here.

Method

The reference solution is the Random Search algorithm which selects a random set of options for each file and provides the benchmark to compare performance of the Simulated Annealing algorithm. The SA algorithm differs from that described in Section II by being initialised to a known valid base state (usually with the value f_{max}). The *distance* property is also modified and becomes the probability that individual file will have its compiler options changed at any iteration. e.g. $d = 0.5$ will result, on average, in 50% of the files having their options changed from the current state.

Unfortunately, it has been not been possible to perform a full exhaustive search of a real application across all possible combinations of options for each file to create a dataset. Instead two synthetic dataset have been generated that try and emulate the behaviour expected in real-life. The two spaces are:

- a small dataset of just eight files, each of which can be set to one of four compiler option sets. Two of the file/option combinations will cause the model to experiment to fail if selected to emulate situations where the executable does not produce the right answer or fails to complete due to numerical instability.
- a second larger dataset, with 1024×5 file/options combinations of which 339 will cause the model to fail if selected (6%).

Each experiments is run 1000 times with limited number of iterations between 4 and 256. The results are recorded and averaged to find the relative mean solution for the particular algorithm/iteration count combination.

Results

Just as in previous sections, the arguments to the algorithm have a significant affect on the performance of the evaluation. Table III describes the arguments used, and Table IV shows the results for each dataset for the Random Search algorithm and Simulated Annealing with two different sets of arguments.

ALGORITHM	PARAMETER SETTINGS
Simulated Annealing d=0.01	$T_{max} = 16.0$, $d = 0.01$, $\lambda_T = 32.0$
Simulated Annealing d=0.64	$T_{max} = 16.0$, $d = 0.64$, $\lambda_T = 32.0$

Table III: Two sets of arguments to the Simulated Annealing algorithm for compiler options

ITERS	RS	SA d=0.64	SA d=0.01
4	0.411	0.431	0.977
8	0.294	0.190	0.962
16	0.210	0.068	0.917

(a) The mean relative solution for 8×4 dataset

ITERS	RS	SA d=0.64	SA d=0.01
4	1.0	1.0	0.988
8	1.0	1.0	0.976
16	1.0	1.0	0.953
32	1.0	1.0	0.901
64	1.0	1.0	0.830
128	1.0	1.0	0.701
256	1.0	1.0	0.518

(b) The mean relative solution 1024×5 dataset

Table IV: The mean relative solutions for compiler flag search

Conclusions

The results show that Simulated Annealing can significantly outperform Random Search with the right settings when allowed more than 8 iterations. However the results also show that if the diameter is too large or small it can severely affect the ability to find any new solutions.

The reason for this becomes clear when considering the number of failure cases in the datasets. If 6% of options will result in a failure then, the probability of randomly selecting 1024 working cases at random is:

$$P = (1 - 0.06)^{1024} = 7.01 \times 10^{-19}$$

Similarly, the probability of finding any solution through Simulated Annealing with $d = 0.64$ becomes:

$$P = (1 - 0.06)^{0.64 \times 1024} = 2.41 \times 10^{-12}$$

If the diameter reduced, such that $d = 0.01$ then the probability of finding any new solution becomes a much more likely:

$$P = (1 - 0.06)^{0.01 \times 1024} = 0.658$$

Therefore it is highly unlikely, with the small number of iterations being evaluated, that a working solution will be discovered by either Random Search or SA $d = 0.64$ on the large data set, let alone one that has better performance.

Instead, the algorithm could be improved, such that P, the probability of finding a new working solution was the tunable argument (rather than d). This would mean increasing or decreasing the value of d depending upon the ratio of failing compiler options, something which is not available to the algorithm before hand, but could be estimated from the number of evaluations, creating an adaptive algorithm.

Either way, Simulated Annealing is still probably the only valid technique for improving the performance of compiler flags, even if the value of d has to be held artificially low for any progress to be made.

ACKNOWLEDGEMENTS

The author kindly acknowledges the contributions of colleagues Dr J. Beech-Brandt, Dr A. Hart and Dr H. Richardson of Cray UK for their assistance in preparing this paper.

This work has been supported by the CRESTA project that has received funding from the European Community's Seventh Framework Programme (ICT-2011.9.13) under Grant Agreement no. 287703.

REFERENCES

[1] M. Giles and G. Mudalige, "Optimising the op2 framework for gpu architectures," in *MRSC*, April 2011.
[2] W. Press, S. Teukolsky, W. Vertterling, and B. Flannery, *Numerical Recipes*. Cambridge University Press, 2007.
[3] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proc. IEEE Int'l Conf. on Neural Networks*, vol. 4, pp. 1942–1948, 1995.
[4] E. Jones, T. Oliphant, P. Peterson, *et al.*, "SciPy: Open source scientific tools for Python."